par Jean-Michel DOUDOUX

Partie 1 : les bases du langage Java
Partie 2 : Le développement des interfaces graphiques
Partie 3 : Les API avancées
Partie 4 : L'utilisation de documents XML
Partie 5 : l'accès aux base de données
Partie 6 : la machine virtuelle Java (JVM)
Partie 7 : Le développement d'applications d'entreprises
Partie 8 : Le développement d'applications web
Partie 9 : Le développement d'applications RIA/RDA
Partie 10 : Les outils pour le développement
Partie 11 : Concevoir et développer des applications
Partie 12 : Le développement d'applications mobiles

Bases du langage POO Packages de bases Exceptions Multitâche

JDK 1.5 Java SE 6.0 Annotations AWT Swing SWT/JFace Applet

Collections Flux Sérialisation Réseau Introspection RMI I18N

Java beans Logging JNI Sécurité Java Web Start JNDI Scripting

JMX XML SAX SAX DOM XSLT JDOM StAX JAXB JDBC JDO

Hibernate J2EE Servlets JSP JSTL JSF Struts Framework web

RIA/RDA Ajax GWT JMS JavaMail EJB Services web UML

Design patterns Outils Normes de développement JavaDoc Ant

Maven Frameworks Tests JUnit Mocks Bibliothèques

J2ME CLDC MIDP CDC

Développons en Java

Jean Michel DOUDOUX

Table des matières

Développons en Java.....1

Préambule.....2
A propos de ce document.....2
Remerciements.....4
Notes de licence.....4
Marques déposées.....4
Historique des versions.....5

Partie 1 : les bases du langage Java.....8

1. Présentation de Java.....9
1.1. Les caractéristiques.....9
1.2. Un bref historique de Java.....10
1.3. Les différentes éditions et versions de Java.....11
1.3.1. Les évolutions des plates-formes Java.....12
1.3.2. Les différentes versions de Java.....12
1.3.3. Java 1.0.....13
1.3.4. Java 1.1.....13
1.3.5. Java 1.2 (nom de code Playground).....14
1.3.6. J2SE 1.3 (nom de code Kestrel).....14
1.3.7. J2SE 1.4 (nom de code Merlin).....14
1.3.8. J2SE 5.0 (nom de code Tiger).....15
1.3.9. Java SE 6 (nom de code Mustang).....15
1.3.10. Java 6 update.....18
1.3.10.1. Java 6 update 1.....19
1.3.10.2. Java 6 update 2.....19
1.3.10.3. Java 6 update 3.....19
1.3.10.4. Java 6 update 4.....19
1.3.10.5. Java 6 update 5.....19
1.3.10.6. Java 6 update 6.....21
1.3.10.7. Java 6 update 7.....21
1.3.10.8. Java 6 update 10.....22
1.3.10.9. Java 6 update 11.....22
1.3.10.10. Java 6 update 12.....22
1.3.10.11. Java 6 update 13.....22
1.3.10.12. Java 6 update 14.....22
1.3.10.13. Java 6 update 15.....23
1.3.10.14. Java 6 update 16.....23
1.3.11. Les futures versions de Java.....23
1.3.12. Le résumé des différentes versions.....23
1.3.13. Les extensions du JDK.....23
1.4. Un rapide tour d'horizon des API et de quelques outils.....24
1.5. Les différences entre Java et JavaScript.....25
1.6. L'installation du JDK.....26
1.6.1. L'installation de la version 1.3 DU JDK de Sun sous Windows 9x.....26
1.6.2. L'installation de la documentation de Java 1.3 sous Windows.....28
1.6.3. La configuration des variables système sous Windows 9x.....29
1.6.4. Les éléments du JDK 1.3 sous Windows.....30
1.6.5. L'installation de la version 1.4.2 du JDK de Sun sous Windows.....30
1.6.6. L'installation de la version 1.5 du JDK de Sun sous Windows.....31
1.6.7. Installation JDK 1.4.2 sous Linux Mandrake 10.....32

2. Les notions et techniques de base en Java.....36
2.1. Les concepts de base.....36
2.1.1. La compilation et l'exécution.....36
2.1.2. Les packages.....39
2.1.3. Le déploiement sous la forme d'un jar.....41
2.1.4. Le classpath.....42

Table des matières

2. Les notions et techniques de base en Java
2.1.4.1. La définition du classpath pour exécuter une application 44
2.1.4.2. La définition du classpath pour exécuter une application avec la variable CLASSPATH 46
2.1.4.3. La définition du classpath pour exécuter une application utilisant une ou plusieurs bibliothèques 46
2.1.4.4. La définition du classpath pour exécuter une application packagée en jar 47
2.2. L'exécution d'une applet 49

3. La syntaxe et les éléments de bases de Java 50
3.1. Les règles de base 50
3.2. Les identificateurs 50
3.3. Les commentaires 51
3.4. La déclaration et l'utilisation de variables 51
3.4.1. La déclaration de variables 51
3.4.2. Les types élémentaires 52
3.4.3. Le format des types élémentaires 53
3.4.4. L'initialisation des variables 54
3.4.5. L'affectation 54
3.4.6. Les comparaisons 55
3.5. Les opérations arithmétiques 55
3.5.1. L'arithmétique entière 56
3.5.2. L'arithmétique en virgule flottante 56
3.5.3. L'incrémentation et la décrémentation 57
3.6. La priorité des opérateurs 58
3.7. Les structures de contrôles 58
3.7.1. Les boucles 58
3.7.2. Les branchements conditionnels 60
3.7.3. Les débranchements 60
3.8. Les tableaux 60
3.8.1. La déclaration des tableaux 61
3.8.2. L'initialisation explicite d'un tableau 61
3.8.3. Le parcours d'un tableau 62
3.9. Les conversions de types 62
3.9.1. La conversion d'un entier int en chaîne de caractère String 63
3.9.2. La conversion d'une chaîne de caractères String en entier int 63
3.9.3. La conversion d'un entier int en entier long 63
3.10. La manipulation des chaînes de caractères 63
3.10.1. Les caractères spéciaux dans les chaines 64
3.10.2. L'addition de chaînes de caractères 64
3.10.3. La comparaison de deux chaines 65
3.10.4. La détermination de la longueur d'une chaine 65
3.10.5. La modification de la casse d'une chaine 65

4. La programmation orientée objet 66
4.1. Le concept de classe 66
4.1.1. La syntaxe de déclaration d'une classe 66
4.2. Les objets 67
4.2.1. La création d'un objet : instancier une classe 67
4.2.2. La durée de vie d'un objet 69
4.2.3. La création d'objets identiques 69
4.2.4. Les références et la comparaison d'objets 69
4.2.5. L'objet null 70
4.2.6. Les variables de classes 70
4.2.7. La variable this 70
4.2.8. L'opérateur instanceof 71
4.3. Les modificateurs d'accès 72
4.3.1. Les mots clés qui gèrent la visibilité des entités 72
4.3.2. Le mot clé static 72
4.3.3. Le mot clé final 73

Table des matières

4. La programmation orientée objet
4.3.4. Le mot clé abstract....74
4.3.5. Le mot clé synchronized....74
4.3.6. Le mot clé volatile....74
4.3.7. Le mot clé native....75
4.4. Les propriétés ou attributs....75
4.4.1. Les variables d'instances....75
4.4.2. Les variables de classes....75
4.4.3. Les constantes....75
4.5. Les méthodes....76
4.5.1. La syntaxe de la déclaration....76
4.5.2. La transmission de paramètres....77
4.5.3. L'émission de messages....78
4.5.4. L'enchainement de références à des variables et à des méthodes....78
4.5.5. La surcharge de méthodes....78
4.5.6. Les constructeurs....79
4.5.7. Le destructeur....80
4.5.8. Les accesseurs....80
4.6. L'héritage....80
4.6.1. Le principe de l'héritage....80
4.6.2. La mise en oeuvre de l'héritage....81
4.6.3. L'accès aux propriétés héritées....81
4.6.4. La redéfinition d'une méthode héritée....81
4.6.5. Le polymorphisme....81
4.6.6. Le transtypage induit par l'héritage facilite le polymorphisme....82
4.6.7. Les interfaces et l'héritage multiple....82
4.6.8. Des conseils sur l'héritage....84
4.7. Les packages....84
4.7.1. La définition d'un package....84
4.7.2. L'utilisation d'un package....84
4.7.3. La collision de classes....85
4.7.4. Les packages et l'environnement système....85
4.8. Les classes internes....85
4.8.1. Les classes internes non statiques....88
4.8.2. Les classes internes locales....92
4.8.3. Les classes internes anonymes....94
4.8.4. Les classes internes statiques....95
4.9. La gestion dynamique des objets....96

5. Les packages de bases....97
5.1. Les packages selon la version du JDK....97
5.2. Le package java.lang....103
5.2.1. La classe Object....103
5.2.1.1. La méthode getClass()....104
5.2.1.2. La méthode toString()....104
5.2.1.3. La méthode equals()....104
5.2.1.4. La méthode finalize()....104
5.2.1.5. La méthode clone()....105
5.2.2. La classe String....105
5.2.3. La classe StringBuffer....107
5.2.4. Les wrappers....108
5.2.5. La classe System....109
5.2.5.1. L'utilisation des flux d'entrée/sortie standard....109
5.2.5.2. Les variables d'environnement et les propriétés du système....111
5.2.6. Les classes Runtime et Process....113
5.3. La présentation rapide du package awt java....116
5.4. La présentation rapide du package java.io....116
5.5. Le package java.util....116
5.5.1. La classe StringTokenizer....116

Table des matières

5. Les packages de bases

5.5.2. La classe Random....117
5.5.3. Les classes Date et Calendar....117
5.5.4. La classe SimpleDateFormat....118
5.5.5. La classe Vector....124
5.5.6. La classe Hashtable....125
5.5.7. L'interface Enumeration....125
5.5.8. La manipulation d'archives zip....126
5.5.9. Les expressions régulières....129
5.5.9.1. Les motifs....130
5.5.9.2. La classe Pattern....131
5.5.9.3. La classe Matcher....132
5.5.10. La classe Formatter....134
5.5.11. La classe Scanner....135
5.6. La présentation rapide du package java.net....136
5.7. La présentation rapide du package java.applet....136

6. Les fonctions mathématiques....137

6.1. Les variables de classe....137
6.2. Les fonctions trigonométriques....138
6.3. Les fonctions de comparaisons....138
6.4. Les arrondis....138
6.4.1. La méthode round(n)....138
6.4.2. La méthode rint(double)....139
6.4.3. La méthode floor(double)....139
6.4.4. La méthode ceil(double)....140
6.4.5. La méthode abs(x)....140
6.5. La méthode IEEEremainder(double, double)....140
6.6. Les Exponentielles et puissances....141
6.6.1. La méthode pow(double, double)....141
6.6.2. La méthode sqrt(double)....141
6.6.3. La méthode exp(double)....141
6.6.4. La méthode log(double)....142
6.7. La génération de nombres aléatoires....142
6.7.1. La méthode random()....142
6.8. La classe BigDecimal....142

7. La gestion des exceptions....149

7.1. Les mots clés try, catch et finally....150
7.2. La classe Throwable....151
7.3. Les classes Exception, RunTimeException et Error....152
7.4. Les exceptions personnalisées....152
7.5. Les exceptions chaînées....153
7.6. L'utilisation des exceptions....154

8. Le multitâche....156

8.1. L'interface Runnable....156
8.2. La classe Thread....157
8.3. La création et l'exécution d'un thread....159
8.3.1. La dérivation de la classe Thread....159
8.3.2. L'implémentation de l'interface Runnable....160
8.3.3. La modification de la priorité d'un thread....161
8.4. La classe ThreadGroup....162
8.5. Un thread en tâche de fond (démon)....162
8.6. L'exclusion mutuelle....163
8.6.1. La sécurisation d'une méthode....163
8.6.2. La sécurisation d'un bloc....163
8.6.3. La sécurisation de variables de classes....164
8.6.4. La synchronisation : les méthodes wait() et notify()....164

Table des matières

9. JDK 1.5 (nom de code Tiger)........165
9.1. Les nouveautés du langage Java version 1.5........165
9.2. L'autoboxing / unboxing........165
9.3. Les importations statiques........166
9.4. Les annotations ou méta données (Meta Data)........167
9.5. Les arguments variables (varargs)........167
9.6. Les generics........169
9.7. Les boucles pour le parcours des collections........172
9.8. Les énumérations (type enum)........174
9.8.1. La définition d'une énumération........174
9.8.2. L'utilisation d'une énumération........176
9.8.3. L'enrichissement de l'énumération........177
9.8.4. La personnalisation de chaque élément........180
9.8.5. Les limitations dans la mise en oeuvre des énumérations........182

10. Les annotations........183
10.1. La mise en oeuvre des annotations........184
10.2. L'utilisation des annotations........185
10.2.1. La documentation........185
10.2.2. L'utilisation par le compilateur........186
10.2.3. La génération de code........186
10.2.4. La génération de fichiers........186
10.2.5. Les API qui utilisent les annotations........186
10.3. Les annotations standards........186
10.3.1. L'annotation @Deprecated........186
10.3.2. L'annotation @Override........187
10.3.3. L'annotation @SuppressWarning........189
10.4. Les annotations communes (Common Annotations)........190
10.4.1. L'annotation @Generated........190
10.4.2. Les annotations @Resource et @Resources........191
10.4.3. Les annotations @PostConstruct et @PreDestroy........192
10.5. Les annotations personnalisées........192
10.5.1. La définition d'une annotation........192
10.5.2. Les annotations pour les annotations........194
10.5.2.1. L'annotation @Target........194
10.5.2.2. L'annotation @Retention........195
10.5.2.3. L'annotation @Documented........195
10.5.2.4. L'annotation @Inherited........196
10.6. L'exploitation des annotations........196
10.6.1. L'exploitation des annotations dans un Doclet........196
10.6.2. L'exploitation des annotations avec l'outil Apt........197
10.6.3. L'exploitation des annotations par introspection........203
10.6.4. L'exploitation des annotations par le compilateur Java........205
10.7. L'API Pluggable Annotation Processing........205
10.7.1. Les processeurs d'annotations........206
10.7.2. L'utilisation des processeurs par le compilateur........208
10.7.3. La création de nouveaux fichiers........208
10.8. Les ressources relatives aux annotations........210

Partie 2 : Développement des interfaces graphiques........211

11. Le graphisme........212
11.1. Les opérations sur le contexte graphique........212
11.1.1. Le tracé de formes géométriques........212
11.1.2. Le tracé de texte........213
11.1.3. L'utilisation des fontes........213
11.1.4. La gestion de la couleur........214
11.1.5. Le chevauchement de figures graphiques........214
11.1.6. L'effacement d'une aire........214

Table des matières

11. Le graphisme
11.1.7. La copier une aire rectangulaire....214

12. Les éléments d'interfaces graphiques de l'AWT....215
12.1. Les composants graphiques....216
12.1.1. Les étiquettes....216
12.1.2. Les boutons....217
12.1.3. Les panneaux....217
12.1.4. Les listes déroulantes (combobox)....218
12.1.5. La classe TextComponent....219
12.1.6. Les champs de texte....219
12.1.7. Les zones de texte multilignes....220
12.1.8. Les listes....222
12.1.9. Les cases à cocher....225
12.1.10. Les boutons radio....226
12.1.11. Les barres de défilement....227
12.1.12. La classe Canvas....228
12.2. La classe Component....229
12.3. Les conteneurs....230
12.3.1. Le conteneur Panel....231
12.3.2. Le conteneur Window....231
12.3.3. Le conteneur Frame....231
12.3.4. Le conteneur Dialog....233
12.4. Les menus....234
12.4.1. Les méthodes de la classe MenuBar....236
12.4.2. Les méthodes de la classe Menu....236
12.4.3. Les méthodes de la classe MenuItem....236
12.4.4. Les méthodes de la classe CheckboxMenuItem....237
12.5. La classe java.awt.Desktop....237

13. La création d'interfaces graphiques avec AWT....239
13.1. Le dimensionnement des composants....239
13.2. Le positionnement des composants....240
13.2.1. La mise en page par flot (FlowLayout)....241
13.2.2. La mise en page bordure (BorderLayout)....242
13.2.3. La mise en page de type carte (CardLayout)....243
13.2.4. La mise en page GridLayout....244
13.2.5. La mise en page GridBagLayout....246
13.3. La création de nouveaux composants à partir de Panel....247
13.4. L'activation ou la désactivation des composants....248

14. L'interception des actions de l'utilisateur....249
14.1. L'interception des actions de l'utilisateur avec Java version 1.0....249
14.2. L'interception des actions de l'utilisateur avec Java version 1.1....249
14.2.1. L'interface ItemListener....251
14.2.2. L'interface TextListener....252
14.2.3. L'interface MouseMotionListener....253
14.2.4. L'interface MouseListener....253
14.2.5. L'interface WindowListener....254
14.2.6. Les différentes implémentations des Listeners....255
14.2.6.1. Une classe implémentant elle même le listener....255
14.2.6.2. Une classe indépendante implémentant le listener....256
14.2.6.3. Une classe interne....257
14.2.6.4. Une classe interne anonyme....257
14.2.7. Résumé....258

15. Le développement d'interfaces graphiques avec SWING....259
15.1. La présentation de Swing....259
15.2. Les packages Swing....260

Table des matières

15. Le développement d'interfaces graphiques avec SWING
15.3. Un exemple de fenêtre autonome......260
15.4. Les composants Swing......261
15.4.1. La classe JFrame......262
15.4.1.1. Le comportement par défaut à la fermeture......264
15.4.1.2. La personnalisation de l'icône......265
15.4.1.3. Centrer une JFrame à l'écran......265
15.4.1.4. Les événements associées à un JFrame......266
15.4.2. Les étiquettes : la classe JLabel......266
15.4.3. Les panneaux : la classe JPanel......269
15.5. Les boutons......269
15.5.1. La classe AbstractButton......269
15.5.2. La classe JButton......271
15.5.3. La classe JToggleButton......272
15.5.4. La classe ButtonGroup......273
15.5.5. Les cases à cocher : la classe JCheckBox......273
15.5.6. Les boutons radio : la classe JRadioButton......274
15.6. Les composants de saisie de texte......274
15.6.1. La classe JTextComponent......275
15.6.2. La classe JTextField......276
15.6.3. La classe JPasswordField......276
15.6.4. La classe JFormattedTextField......278
15.6.5. La classe JEditorPane......278
15.6.6. La classe JTextPane......279
15.6.7. La classe JTextArea......279
15.7. Les onglets......281
15.8. Le composant JTree......282
15.8.1. La création d'une instance de la classe JTree......282
15.8.2. La gestion des données de l'arbre......285
15.8.2.1. L'interface TreeNode......286
15.8.2.2. L'interface MutableTreeNode......286
15.8.2.3. La classe DefaultMutableTreeNode......287
15.8.3. La modification du contenu de l'arbre......288
15.8.3.1. Les modifications des noeuds fils......288
15.8.3.2. Les événements émis par le modèle......289
15.8.3.3. L'édition d'un noeud......290
15.8.3.4. Les éditeurs personnalisés......290
15.8.3.5. 3.5 La définition des noeuds éditables......291
15.8.4. La mise en oeuvre d'actions sur l'arbre......292
15.8.4.1. Etendre ou refermer un noeud......292
15.8.4.2. La détermination du noeud sélectionné......293
15.8.4.3. Le parcours des noeuds de l'arbre......293
15.8.5. La gestion des événements......294
15.8.5.1. La classe TreePath......295
15.8.5.2. La gestion de la sélection d'un noeud......296
15.8.5.3. Les événements lies à la sélection de noeuds......297
15.8.5.4. Les événements lorsqu'un noeud est étendu ou refermé......299
15.8.5.5. Le contrôle des actions pour étendre ou refermer un noeud......300
15.8.6. La personnalisation du rendu......300
15.8.6.1. Personnaliser le rendu des noeuds......301
15.8.6.2. Les bulles d'aides (Tooltips)......304
15.9. Les menus......304
15.9.1. La classe JMenuBar......307
15.9.2. La classe JMenuItem......309
15.9.3. La classe JPopupMenu......309
15.9.4. La classe JMenu......311
15.9.5. La classe JCheckBoxMenuItem......312
15.9.6. La classe JRadioButtonMenuItem......312
15.9.7. La classe JSeparator......312

Table des matières

15. Le développement d'interfaces graphiques avec SWING
15.10. L'affichage d'une image dans une application....313

16. Le développement d'interfaces graphiques avec SWT....318
16.1. La présentation de SWT....318
16.2. Un exemple très simple....320
16.3. La classe SWT....320
16.4. L'objet Display....321
16.5. L'objet Shell....321
16.6. Les composants....323
16.6.1. La classe Control....323
16.6.2. Les contrôles de base....324
16.6.2.1. La classe Button....324
16.6.2.2. La classe Label....324
16.6.2.3. La classe Text....325
16.6.3. Les contrôles de type liste....326
16.6.3.1. La classe Combo....326
16.6.3.2. La classe List....327
16.6.4. Les contrôles pour les menus....327
16.6.4.1. La classe Menu....327
16.6.4.2. La classe MenuItem....328
16.6.5. Les contrôles de sélection ou d'affichage d'une valeur....329
16.6.5.1. La classe ProgressBar....329
16.6.5.2. La classe Scale....330
16.6.5.3. La classe Slider....330
16.6.6. Les contrôles de type « onglets »....331
16.6.6.1. La classe TabFolder....331
16.6.6.2. La classe TabItem....332
16.6.7. Les contrôles de type « tableau »....332
16.6.7.1. La classe Table....332
16.6.7.2. La classe TableColumn....334
16.6.7.3. La classe TableItem....334
16.6.8. Les contrôles de type « arbre »....335
16.6.8.1. La classe Tree....335
16.6.8.2. La classe TreeItem....336
16.6.9. La classe ScrollBar....336
16.6.10. Les contrôles pour le graphisme....336
16.6.10.1. La classe Canvas....336
16.6.10.2. La classe GC....337
16.6.10.3. La classe Color....338
16.6.10.4. La classe Font....338
16.6.10.5. La classe Image....339
16.7. Les conteneurs....340
16.7.1. Les conteneurs de base....340
16.7.1.1. La classe Composite....340
16.7.1.2. La classe Group....342
16.7.2. Les contrôles de type « barre d'outils »....342
16.7.2.1. La classe ToolBar....342
16.7.2.2. La classe ToolItem....343
16.7.2.3. Les classes CoolBar et Cooltem....345
16.8. La gestion des erreurs....346
16.9. Le positionnement des contrôles....346
16.9.1. Le positionnement absolu....347
16.9.2. Le positionnement relatif avec les LayoutManager....347
16.9.2.1. FillLayout....347
16.9.2.2. RowLayout....348
16.9.2.3. GridLayout....349
16.9.2.4. FormLayout....352
16.10. La gestion des événements....352

Table des matières

16. Le développement d'interfaces graphiques avec SWT
16.10.1. L'interface KeyListener....353
16.10.2. L'interface MouseListener....355
16.10.3. L'interface MouseMoveListener....356
16.10.4. L'interface MouseTrackListener....357
16.10.5. L'interface ModifyListener....357
16.10.6. L'interface VerifyText()....358
16.10.7. L'interface FocusListener....359
16.10.8. L'interface TraverseListener....360
16.10.9. L'interface PaintListener....361
16.11. Les boîtes de dialogue....362
16.11.1. Les boîtes de dialogues prédéfinies....362
16.11.1.1. La classe MessageBox....362
16.11.1.2. La classe ColorDialog....363
16.11.1.3. La classe FontDialog....364
16.11.1.4. La classe FileDialog....365
16.11.1.5. La classe DirectoryDialog....365
16.11.1.6. La classe PrintDialog....366
16.11.2. Les boites de dialogues personnalisées....367

17. JFace....369
17.1. La structure générale d'une application....370
17.2. Les boites de dialogues....371
17.2.1. L'affichage des messages d'erreur....371
17.2.2. L'affichage des messages d'information à l'utilisateur....372
17.2.3. La saisie d'une valeur par l'utilisateur....374
17.2.4. La boîte de dialogue pour afficher la progression d'un traitement....375

Partie 3 : Les API avancées....379

18. Les collections....381
18.1. Présentation du framework collection....381
18.2. Les interfaces des collections....382
18.2.1. L'interface Collection....383
18.2.2. L'interface Iterator....384
18.3. Les listes....385
18.3.1. L'interface List....385
18.3.2. Les listes chaînées : la classe LinkedList....385
18.3.3. L'interface ListIterator....387
18.3.4. Les tableaux redimensionnables : la classe ArrayList....387
18.4. Les ensembles....388
18.4.1. L'interface Set....388
18.4.2. L'interface SortedSet....388
18.4.3. La classe HashSet....389
18.4.4. La classe TreeSet....389
18.5. Les collections gérées sous la forme clé/valeur....390
18.5.1. L'interface Map....390
18.5.2. L'interface SortedMap....391
18.5.3. La classe Hashtable....391
18.5.4. La classe TreeMap....392
18.5.5. La classe HashMap....392
18.6. Le tri des collections....393
18.6.1. L'interface Comparable....393
18.6.2. L'interface Comparator....393
18.7. Les algorithmes....393
18.8. Les exceptions du framework....395

Table des matières

19. Les flux.....396
19.1. La présentation des flux.....396
19.2. Les classes de gestion des flux.....396
19.3. Les flux de caractères.....398
19.3.1. La classe Reader.....399
19.3.2. La classe Writer.....400
19.3.3. Les flux de caractères avec un fichier.....400
19.3.3.1. Les flux de caractères en lecture sur un fichier.....400
19.3.3.2. Les flux de caractères en écriture sur un fichier.....401
19.3.4. Les flux de caractères tamponnés avec un fichier......401
19.3.4.1. Les flux de caractères tamponnés en lecture avec un fichier.....401
19.3.4.2. Les flux de caractères tamponnés en écriture avec un fichier.....402
19.3.4.3. La classe PrintWriter.....403
19.4. Les flux d'octets.....405
19.4.1. Les flux d'octets avec un fichier......405
19.4.1.1. Les flux d'octets en lecture sur un fichier.....405
19.4.1.2. Les flux d'octets en écriture sur un fichier.....406
19.4.2. Les flux d'octets tamponnés avec un fichier......407
19.5. La classe File.....408
19.6. Les fichiers à accès direct.....410
19.7. La classe java.io.Console.....412

20. La sérialisation.....413
20.1. Les classes et les interfaces de la sérialisation.....413
20.1.1. L'interface Serializable.....413
20.1.2. La classe ObjectOuputStream.....414
20.1.3. La classe ObjectInputStream.....415
20.2. Le mot clé transient.....416
20.3. La sérialisation personnalisée.....417
20.3.1. L'interface Externalizable.....417

21. L'interaction avec le réseau.....418
21.1. L'introduction aux concepts liés au réseau.....418
21.2. Les adresses internet.....419
21.2.1. La classe InetAddress.....419
21.3. L'accès aux ressources avec une URL.....420
21.3.1. La classe URL.....420
21.3.2. La classe URLConnection.....421
21.3.3. La classe URLEncoder.....422
21.3.4. La classe HttpURLConnection.....423
21.4. L'utilisation du protocole TCP.....423
21.4.1. La classe SocketServer.....424
21.4.2. La classe Socket.....426
21.5. L'utilisation du protocole UDP.....427
21.5.1. La classe DatagramSocket.....427
21.5.2. La classe DatagramPacket.....428
21.5.3. Un exemple de serveur et de client.....428
21.6. Les exceptions liées au réseau.....430
21.7. Les interfaces de connexions au réseau.....430

22. La gestion dynamique des objets et l'introspection.....432
22.1. La classe Class.....432
22.1.1. L'obtention d'un objet à partir de la classe Class.....433
22.1.1.1. La détermination de la classe d'un objet.....433
22.1.1.2. L'obtention d'un objet Class à partir d'un nom de classe.....433
22.1.1.3. Une troisième façon d'obtenir un objet Class.....434
22.1.2. Les méthodes de la classe Class.....434
22.2. La recherche des informations sur une classe.....435
22.2.1. La recherche de la classe mère d'une classe.....435

Table des matières

22. La gestion dynamique des objets et l'introspection
22.2.2. La recherche des modificateurs d'une classe....435
22.2.3. La recherche des interfaces implémentées par une classe....436
22.2.4. La recherche des champs publics....436
22.2.5. La recherche des paramètres d'une méthode ou d'un constructeur....437
22.2.6. La recherche des constructeurs de la classe....438
22.2.7. La recherche des méthodes publiques....439
22.2.8. La recherche de toutes les méthodes....440
22.3. La définition dynamique d'objets....440
22.3.1. La définition d'objets grâce à la classe Class....440
22.3.2. L'exécution dynamique d'une méthode....440

23. L'appel de méthodes distantes : RMI....442
23.1. La présentation et l'architecture de RMI....442
23.2. Les différentes étapes pour créer un objet distant et l'appeler avec RMI....442
23.3. Le développement coté serveur....443
23.3.1. La définition d'une interface qui contient les méthodes de l'objet distant....443
23.3.2. L'écriture d'une classe qui implémente cette interface....443
23.3.3. L'écriture d'une classe pour instancier l'objet et l'enregistrer dans le registre....444
23.3.3.1. La mise en place d'un security manager....444
23.3.3.2. L'instanciation d'un objet de la classe distante....444
23.3.3.3. L'enregistrement dans le registre de nom RMI en lui donnant un nom....445
23.3.3.4. Le lancement dynamique du registre de nom RMI....445
23.4. Le développement coté client....446
23.4.1. La mise en place d'un security manager....446
23.4.2. L'obtention d'une référence sur l'objet distant à partir de son nom....446
23.4.3. L'appel à la méthode à partir de la référence sur l'objet distant....447
23.4.4. L'appel d'une méthode distante dans une applet....447
23.5. La génération des classes stub et skeleton....448
23.6. La mise en oeuvre des objets RMI....448
23.6.1. Le lancement du registre RMI....448
23.6.2. L'instanciation et l'enregistrement de l'objet distant....449
23.6.3. Le lancement de l'application cliente....449

24. L'internationalisation....450
24.1. Les objets de type Locale....450
24.1.1. La création d'un objet Locale....450
24.1.2. L'obtention de la liste des Locales disponibles....451
24.1.3. L'utilisation d'un objet Locale....452
24.2. La classe ResourceBundle....452
24.2.1. La création d'un objet ResourceBundle....452
24.2.2. Les sous classes de ResourceBundle....453
24.2.2.1. L'utilisation de PropertyResourceBundle....453
24.2.2.2. L'utilisation de ListResourceBundle....453
24.2.3. L'obtention d'un texte d'un objet ResourceBundle....454
24.3. Des chemins guidés pour réaliser la localisation....454
24.3.1. L'utilisation d'un ResourceBundle avec un fichier propriétés....454
24.3.2. Des exemples de classes utilisant PropertiesResourceBundle....455
24.3.3. L'utilisation de la classe ListResourceBundle....456
24.3.4. Des exemples de classes utilisant ListResourceBundle....457
24.3.5. La création de sa propre classe fille de ResourceBundle....459

25. Les composants Java beans....463
25.1. La présentation des Java beans....463
25.2. Les propriétés....464
25.2.1. Les propriétés simples....464
25.2.2. Les propriétés indexées (indexed properties)....465
25.2.3. Les propriétés liées (Bound properties)....465
25.2.4. Les propriétés liées avec contraintes (Constrained properties)....467

Table des matières

25. Les composants Java beans
25.3. Les méthodes....469
25.4. Les événements....469
25.5. L'introspection....469
25.5.1. Les modèles (designs patterns)....470
25.5.2. La classe BeanInfo....470
25.6. Paramétrage du bean (Customization)....472
25.7. La persistance....472
25.8. La diffusion sous forme de jar....472

26. Logging....474
26.1. La présentation du logging....474
26.1.1. Des recommandations lors de la mise en oeuvre....475
26.1.2. Les différents frameworks....475
26.2. Log4j....476
26.2.1. Les premiers pas....477
26.2.1.1. L'installation....477
26.2.1.2. Les principes de mise en oeuvre....478
26.2.1.3. Un exemple de mise en oeuvre....478
26.2.2. La gestion des logs avec les versions antérieures à la 1.2....479
26.2.2.1. Les niveaux de gravités : la classe Priority....479
26.2.2.2. La classe Category....480
26.2.2.3. La hiérarchie dans les Categorys....481
26.2.3. La gestion des logs à partir de la version 1.2....482
26.2.3.1. Les niveaux de gravité : la classe Level....482
26.2.3.2. La classe Logger....482
26.2.3.3. La migration de Log4j antérieure à 1.2 vers 1.2....484
26.2.4. Les Appenders....485
26.2.4.1. AsyncAppender....486
26.2.4.2. JDBCAppender....487
26.2.4.3. JMSAppender....488
26.2.4.4. LF5Appender....488
26.2.4.5. NTEventLogAppender....488
26.2.4.6. NullAppender....488
26.2.4.7. SMTPAppender....488
26.2.4.8. SocketAppender....489
26.2.4.9. SocketHubAppender....490
26.2.4.10. SyslogAppender....490
26.2.4.11. TelnetAppender....490
26.2.4.12. WriterAppender....490
26.2.4.13. ConsoleAppender....491
26.2.4.14. FileAppender....492
26.2.4.15. DailyRollingFileAppender....493
26.2.4.16. RollingFileAppender....494
26.2.4.17. ExternalyRolledFileAppender....494
26.2.5. Les layouts....495
26.2.5.1. SimpleLayout....495
26.2.5.2. HTMLLayout....496
26.2.5.3. XMLLayout....497
26.2.5.4. PatternLayout....497
26.2.6. L'externalisation de la configuration....500
26.2.6.1. Les principes généraux....500
26.2.6.2. Le chargement explicite d'une configuration....501
26.2.6.3. Les formats des fichiers de configuration....504
26.2.6.4. La configuration par fichier properties....504
26.2.6.5. La configuration via un fichier XML....506
26.2.6.6. log4j.xml versus log4j.properties....513
26.2.6.7. La conversion du format properties en format XML....513
26.2.7. La mise en oeuvre avancée....514

Table des matières

26. Logging

26.2.7.1. La lecture des logs....514
26.2.7.2. Les variables d'environnement....515
26.2.7.3. L'internationalisation des messages....516
26.2.7.4. L'initialisation de Log4j dans une webapp....517
26.2.7.5. La modification dynamique de la configuration....517
26.2.7.6. NDC/MDC....517
26.2.8. Des best practices....517
26.2.8.1. Le choix du niveau de gravité des messages....517
26.2.8.2. L'amélioration des performances....518
26.3. L'API logging....519
26.3.1. La classe LogManager....519
26.3.2. La classe Logger....520
26.3.3. La classe Level....520
26.3.4. La classe LogRecord....520
26.3.5. La classe Handler....520
26.3.6. La classe Filter....521
26.3.7. La classe Formatter....521
26.3.8. Le fichier de configuration....521
26.3.9. Exemples d'utilisation....522
26.4. Jakarta Commons Logging (JCL)....522
26.5. D'autres API de logging....523

27. La sécurité....524
27.1. La sécurité dans les spécifications du langage....524
27.1.1. Les contrôles lors de la compilation....524
27.1.2. Les contrôles lors de l'exécution....525
27.2. Le contrôle des droits d'une application....525
27.2.1. Le modèle de sécurité de Java 1.0....525
27.2.2. Le modèle de sécurité de Java 1.1....525
27.2.3. Le modèle Java 1.2....525
27.3. JCE (Java Cryptography Extension)....526
27.3.1. La classe Cipher....526
27.4. JSSE (Java Secure Sockets Extension)....526
27.5. JAAS (Java Authentication and Authorization Service)....526

28. JNI (Java Native Interface)....527
28.1. La déclaration et l'utilisation d'une méthode native....527
28.2. La génération du fichier d'en-tête....528
28.3. L'écriture du code natif en C....529
28.4. Le passage de paramètres et le renvoi d'une valeur (type primitif)....531
28.5. Le passage de paramètres et le renvoi d'une valeur (type objet)....532

29. JNDI (Java Naming and Directory Interface)....535
29.1. La présentation de JNDI....535
29.1.1. Les services de nommage....536
29.1.2. Les annuaires....537
29.1.3. Le contexte....537
29.2. La mise en oeuvre de l'API JNDI....538
29.2.1. L'interface Name....538
29.2.2. L'interface Context et la classe IntitialContext....538
29.3. L'utilisation d'un service de nommage....539
29.3.1. L'obtention d'un objet....540
29.3.2. Le stockage d'un objet....540
29.4. L'utilisation avec un DNS....540
29.5. L'utilisation du File System Context Provider de Sun....541
29.6. LDAP....542
29.6.1. L'outil OpenLDAP....544
29.6.2. LDAPBrowser....545

Table des matières

29. JNDI (Java Naming and Directory Interface)
29.6.3. LDIF....548
29.7. L'utilisation avec un annuaire LDAP....548
29.7.1. L'interface DirContext....548
29.7.2. La classe InitialDirContext....549
29.7.3. Les attributs....551
29.7.4. L'utilisation d'objets Java....551
29.7.5. Le stockage d'objets Java....551
29.7.6. L'obtention d'un objet Java....553
29.7.7. La modification d'un objet....554
29.7.8. La suppression d'un objet....559
29.7.9. La recherche d'associations....560
29.7.10. La recherche dans un annuaire LDAP....561
29.8. JNDI et J2EE....566

30. Scripting....567
30.1. L'API Scripting....567
30.1.1. La mise en oeuvre de l'API....567
30.1.2. Ajouter d'autres moteurs de scripting....568
30.1.3. L'évaluation d'un script....570
30.1.4. L'interface Compilable....572
30.1.5. L'interface Invocable....573
30.1.6. La commande jrunscript....575

31. JMX (Java Management Extensions)....576
31.1. La présentation de JMX....576
31.2. L'architecture de JMX....578
31.3. Un premier exemple....580
31.3.1. La définition de l'interface et des classes du MBean....580
31.3.2. L'exécution de l'application....581
31.4. La couche instrumentation : les MBeans....584
31.4.1. Les MBeans....584
31.4.2. Les différents types de MBeans....585
31.4.3. Les MBeans dans l'architecture JMX....586
31.4.4. Le nom des MBeans....586
31.4.5. Les types de données dans les MBeans....587
31.4.5.1. Les types de données complexes....588
31.5. Les MBeans standards....588
31.5.1. La définition de l'interface d'un MBean standard....588
31.5.2. L'implémentation du MBean Standard....590
31.5.3. L'utilisation d'un MBean....590
31.6. La couche agent....591
31.6.1. Le rôle d'un agent JMX....591
31.6.2. Le serveur de MBeans (MBean Server)....592
31.6.3. Le Mbean de type MBeanServerDelegate....592
31.6.3.1. L'enregistrement d'un MBean dans le serveur de MBeans....593
31.6.3.2. L'interface MBeanRegistration....593
31.6.3.3. La suppression d'un MBean du serveur de MBeans....593
31.6.4. La communication avec la couche agent....593
31.6.5. Le développement d'un agent JMX....594
31.6.5.1. L'instanciation d'un serveur de MBeans....594
31.6.5.2. L'instanciation et l'enregistrement d'un MBean dans le serveur....595
31.6.5.3. L'ajout d'un connecteur ou d'un adaptateur de protocoles....596
31.6.5.4. L'utilisation d'un service de l'agent....596
31.7. Les services d'un agent JMX....596
31.7.1. Le service de type M-Let....597
31.7.1.1. Le format du fichier de définitions....597
31.7.1.2. L'instanciation et l'utilisation d'un service M-Let dans un agent....598
31.7.1.3. Un exemple de mise en oeuvre du service M-Let....598

Table des matières

31. JMX (Java Management Extensions)
31.7.2. Le service de type Timer....601
31.7.2.1. Les fonctionnalités du service Timer....601
31.7.2.2. L'ajout d'une définition de notifications....602
31.7.2.3. Un exemple de mise en oeuvre du service Timer....602
31.7.3. Le service de type Monitor....604
31.7.4. Le service de type Relation....604
31.8. La couche services distribués....604
31.8.1. L'interface MBeanServerConnection....604
31.8.2. Les connecteurs et les adaptateurs de protocoles....605
31.8.2.1. Les connecteurs....605
31.8.2.2. Les adaptateurs de protocoles....606
31.8.3. L'utilisation du connecteur RMI....606
31.8.4. L'utilisation du connecteur utilisant le protocole JMXMP....610
31.8.5. L'utilisation de l'adaptateur de protocole HTML....613
31.8.6. L'invocation d'un MBean via un proxy....617
31.8.7. La recherche et la découverte des agents JMX....620
31.8.7.1. Via le Service Location Protocol (SLP)....620
31.8.7.2. Via la technologie Jini....620
31.8.7.3. Via un annuaire et la technologie JNDI....620
31.9. Les notifications....620
31.9.1. L'interface NotificationBroadcaster....621
31.9.2. L'interface NotificationEmitter....621
31.9.3. La classe NotificationBroadcasterSupport....621
31.9.4. La classe javax.management.Notification....621
31.9.5. Un exemple de notifications....622
31.9.6. L'abonnement aux notifications par un client JMX....624
31.10. Les Dynamic MBeans....625
31.10.1. L'interface DynamicMBean....625
31.10.2. Les méta données d'un Dynamic MBean....626
31.10.2.1. La classe MBeanInfo....626
31.10.2.2. La classe MBeanFeatureInfo....626
31.10.2.3. La classe MBeanAttributeInfo....627
31.10.2.4. La classe MBeanParameterInfo....627
31.10.2.5. La classe MBeanConstructorInfo....627
31.10.2.6. La classe MBeanOperationInfo....628
31.10.2.7. La classe MBeanNotificationInfo....628
31.10.3. La définition d'un MBean Dynamic....629
31.10.4. La classe StandardMBean....633
31.11. Les Model MBeans....634
31.11.1. L'interface ModelMBean et la classe RequiredModelMBean....634
31.11.2. La description des fonctionnalités exposées....635
31.11.3. Un exemple de mise en oeuvre....636
31.11.4. Les fonctionnalités optionnelles des Model MBeans....639
31.11.5. Les différences entre un Dynamic MBean est un Model MBean....639
31.12. Les Open MBeans....640
31.12.1. La mise en oeuvre d'un Open MBean....640
31.12.2. Les types de données utilisables dans les Open MBeans....641
31.12.2.1. Les Open Types....641
31.12.2.2. La classe CompositeType et l'interface CompositeData....642
31.12.2.3. La classe TabularType et l'interface TabularData....642
31.12.3. Un exemple d'utilisation d'un Open MBean....643
31.12.4. Les avantages et les inconvénients des Open MBeans....643
31.13. Les MXBeans....643
31.13.1. La définition d'un MXBean....643
31.13.2. L'écriture d'un type personnalisé utilisé par le MXBean....644
31.13.3. La mise en oeuvre d'un MXBean....645
31.14. L'interface PersistentMBean....647
31.15. Le monitoring d'une JVM....648

Table des matières

31. JMX (Java Management Extensions)
31.15.1. L'interface ClassLoadingMXBean....648
31.15.2. L'interface CompilationMXBean....649
31.15.3. L'interface GarbageCollectorMXBean....650
31.15.4. L'interface MemoryManagerMXBean....651
31.15.5. L'interface MemoryMXBean....652
31.15.6. L'interface MemoryPoolMXBean....655
31.15.7. L'interface OperatingSystemMXBean....656
31.15.8. L'interface RuntimeMXBean....657
31.15.9. L'interface ThreadMXBean....658
31.15.10. La sécurisation des accès à l'agent....660
31.16. Des recommandations pour l'utilisation de JMX....660
31.17. Des ressources....660

Partie 4 : l'utilisation de documents XML....662

32. Java et XML....663
32.1. La présentation de XML....663
32.2. Les règles pour formater un document XML....663
32.3. La DTD (Document Type Definition)....664
32.4. Les parseurs....664
32.5. La génération de données au format XML....665
32.6. JAXP : Java API for XML Parsing....666
32.6.1. JAXP 1.1....666
32.6.2. L'utilisation de JAXP avec un parseur de type SAX....667
32.7. Jaxen....667

33. SAX (Simple API for XML)....669
33.1. L'utilisation de SAX....669
33.1.1. L'utilisation de SAX de type 1....669
33.1.2. L'utilisation de SAX de type 2....676

34. DOM (Document Object Model)....678
34.1. Les interfaces du DOM....679
34.1.1. L'interface Node....679
34.1.2. L'interface NodeList....680
34.1.3. L'interface Document....680
34.1.4. L'interface Element....680
34.1.5. L'interface CharacterData....681
34.1.6. L'interface Attr....681
34.1.7. L'interface Comment....681
34.1.8. L'interface Text....681
34.2. L'obtention d'un arbre DOM....682
34.3. Le parcours d'un arbre DOM....683
34.3.1. Les interfaces Traversal....683
34.4. La modification d'un arbre DOM....683
34.4.1. La création d'un document....683
34.4.2. L'ajout d'un élément....684
34.5. L'envoie d'un arbre DOM dans un flux....685
34.5.1. Un exemple avec Xerces....685

35. XSLT (Extensible Stylesheet Language Transformations)....687
35.1. XPath....687
35.2. La syntaxe de XSLT....688
35.3. Un exemple avec Internet Explorer....689
35.4. Un exemple avec Xalan 2....689

Table des matières

36. Les modèles de document....691
36.1. L'API JDOM....691
36.1.1. L'historique de JDOM....691
36.1.2. La présentation de JDOM....691
36.1.3. Les fonctionnalités et les caractéristiques....692
36.1.4. L'installation de JDOM....693
36.1.4.1. L'installation de JDOM Betâ 7 sous Windows....693
36.1.4.2. L'installation de la version 1.x 694
36.1.5. Les différentes entités de JDOM....694
36.1.5.1. La classe Document....695
36.1.5.2. La classe DocType....696
36.1.5.3. La classe Element....698
36.1.5.4. La classe Attribut....701
36.1.5.5. La classe Text....705
36.1.5.6. La classe Comment....707
36.1.5.7. La classe Namespace....709
36.1.5.8. La classe CData....712
36.1.5.9. La classe ProcessingInstruction....712
36.1.6. La création d'un document....715
36.1.6.1. La création d'un nouveau document....715
36.1.6.2. L'obtention d'une instance de Document à partir d'un document XML....716
36.1.6.3. La création d'éléments....720
36.1.6.4. L'ajout d'éléments fils....720
36.1.7. L'arborescence d'éléments....722
36.1.7.1. Le parcours des éléments....723
36.1.7.2. L'accès direct à un élément fils....724
36.1.7.3. Le parcours de toute l'arborescence d'un document....727
36.1.7.4. Les éléments parents....730
36.1.8. La modification d'un document....731
36.1.8.1. L'obtention du texte d'un élément....735
36.1.8.2. La modification du texte d'un élément....737
36.1.8.3. L'obtention du texte d'un élément fils....738
36.1.8.4. L'ajout et la suppression des fils....739
36.1.8.5. Le déplacement d'un ou des éléments....741
36.1.8.6. La duplication d'un élément....743
36.1.9. L'utilisation de filtres....745
36.1.10. L'exportation d'un document....748
36.1.10.1. L'exportation dans un flux....748
36.1.10.2. L'exportation dans un arbre DOM....753
36.1.10.3. L'exportation en SAX....753
36.1.11. L'utilisation de XSLT....755
36.1.12. L'utilisation de XPath....758
36.1.13. L'intégration à Java....760
36.1.14. Les contraintes de la mise en oeuvre de JDOM....761
36.2. dom4j....761
36.2.1. L'installation de dom4j....761
36.2.2. La création d'un document....761
36.2.3. Le parcours d'un document....762
36.2.4. La modification d'un document XML....763
36.2.5. La création d'un nouveau document XML....763
36.2.6. L'exportation d'un document....764

37. JAXB (Java Architecture for XML Binding)....767
37.1. JAXB 1.0....767
37.1.1. La génération des classes....768
37.1.2. L'API JAXB....770
37.1.3. L'utilisation des classes générées et de l'API....770
37.1.4. La création d'un nouveau document XML....771
37.1.5. La génération d'un document XML....772

Table des matières

37. JAXB (Java Architecture for XML Binding)
37.2. JAXB 2.0.....773
37.2.1. L'obtention de JAXB 2.0.....776
37.2.2. La mise en oeuvre de JAXB 2.0.....776
37.2.3. La génération des classes à partir d'un schéma.....777
37.2.4. La commande xjc.....778
37.2.5. Les classes générées.....778
37.2.6. L'utilisation de l'API JAXB 2.0.....780
37.2.6.1. Le mapping d'un document XML à des objets (unmarshal).....780
37.2.6.2. La création d'un document XML à partir d'objets.....782
37.2.6.3. En utilisant des classes annotées.....783
37.2.6.4. En utilisant les classes générées à partir d'un schéma.....786
37.2.7. La configuration de la liaison XML / Objets.....787
37.2.7.1. l'annotation du schéma XML.....787
37.2.7.2. L'annotation des classes.....789
37.2.7.3. La génération d'un schéma à partir de classes compilées.....793

38. StAX (Streaming Api for XML).....794
38.1. La présentation de StAX.....794
38.2. Les deux API de StAX.....795
38.3. Les fabriques.....796
38.4. Le traitement d'un document XML avec l'API du type curseur.....798
38.5. Le traitement d'un document XML avec l'API du type itérateur.....805
38.6. La mise en oeuvre des filtres.....808
38.7. L'écriture un document XML avec l'API de type curseur.....811
38.8. L'écriture un document XML avec l'API de type itérateur.....815
38.9. La comparaison entre SAX, DOM et StAX.....818

Partie 5 : L'accès aux bases de données.....821

39. La persistance des objets.....822
39.1. Introduction.....822
39.1.1. La correspondance entre le modèle relationnel et objet.....822
39.2. L'évolution des solutions de persistance avec Java.....822
39.3. Le mapping O/R (objet/relationnel).....823
39.3.1. Le choix d'une solution de mapping O/R.....823
39.3.2. Les difficultés lors de la mise en place d'un outil de mapping O/R.....824
39.4. L'architecture et la persistance de données.....824
39.4.1. La couche de persistance.....824
39.4.2. Les opérations de type CRUD.....825
39.4.3. Le modèle de conception DAO (Data Access Object).....825
39.5. Les différentes solutions.....826
39.6. Les API standards.....826
39.6.1. JDBC.....826
39.6.2. JDO 1.0.....826
39.6.3. JD0 2.0.....827
39.6.4. EJB 2.0.....827
39.6.5. Java Persistence API et EJB 3.0.....827
39.7. Les frameworks open source.....828
39.7.1. iBatis.....828
39.7.2. Hibernate.....828
39.7.3. Castor.....828
39.7.4. Apache Torque.....828
39.7.5. TopLink.....829
39.7.6. Apache OJB.....829
39.7.7. Apache Cayenne.....829
39.8. L'utilisation de procédures stockées.....829

Table des matières

40. JDBC (Java DataBase Connectivity)....830
40.1. Les outils nécessaires pour utiliser JDBC....830
40.2. Les types de pilotes JDBC....830
40.3. L'enregistrement d'une base de données dans ODBC sous Windows 9x ou XP....831
40.4. La présentation des classes de l'API JDBC....833
40.5. La connexion à une base de données....833
40.5.1. Le chargement du pilote....834
40.5.2. L'établissement de la connexion....834
40.6. L'accès à la base de données....835
40.6.1. L'exécution de requêtes SQL....835
40.6.2. La classe ResultSet....837
40.6.3. Un exemple complet de mise à jour et de sélection sur une table....838
40.7. L'obtention d'informations sur la base de données....839
40.7.1. La classe ResultSetMetaData....839
40.7.2. La classe DatabaseMetaData....840
40.8. L'utilisation d'un objet PreparedStatement....841
40.9. L'utilisation des transactions....842
40.10. Les procédures stockées....842
40.11. Le traitement des erreurs JDBC....843
40.12. JDBC 2.0....844
40.12.1. Les fonctionnalités de l'objet ResultSet....845
40.12.2. Les mises à jour de masse (Batch Updates)....847
40.12.3. Le package javax.sql....848
40.12.4. La classe DataSource....848
40.12.5. Les pools de connexion....848
40.12.6. Les transactions distribuées....849
40.12.7. L'API RowSet....849
40.12.7.1. L'interface RowSet....850
40.12.7.2. L'interface JdbcRowSet....851
40.12.7.3. L'interface CachedRowSet....854
40.12.7.4. L'interface WebRowSet....859
40.12.7.5. L'interface FilteredRowSet....862
40.12.7.6. L'interface JoinRowSet....864
40.12.7.7. L'utilisation des événements....865
40.13. JDBC 3.0....866
40.13.1. Le nommage des paramètres d'un objet de type CallableStament....867
40.13.2. Les types java.sql.Types.DATALINK et java.sql.Types.BOOLEAN....867
40.13.3. L'obtention des valeurs générées automatiquement lors d'une insertion....867
40.13.4. Le support des points de sauvegarde (savepoint)....869
40.13.5. Le pool d'objets PreparedStatements....870
40.13.6. La définition de propriétés pour les pools de connexions....870
40.13.7. L'ajout de metadata pour obtenir la liste des types de données supportés....871
40.13.8. La possibilité d'avoir plusieurs ResultSet retournés par un CallableStatement ouverts en même temps....871
40.13.9. Préciser si un ResultSet doit être maintenu ouvert ou fermé à la fin d'une transaction....872
40.13.10. La mise à jour des données de type BLOB, CLOB, REF et ARRAY....872
40.14. MySQL et Java....874
40.14.1. L'installation de MySQL 3.23 sous Windows....874
40.14.2. Les opérations de base avec MySQL....875
40.14.3. L'utilisation de MySQL avec Java via ODBC....877
40.14.3.1. La déclaration d'une source de données ODBC vers la base de données....877
40.14.3.2. L'utilisation de la source de données....879
40.14.4. L'utilisation de MySQL avec Java via un pilote JDBC....880
40.15. L'amélioration des performances avec JDBC....882
40.15.1. Le choix du pilote JDBC à utiliser....882
40.15.2. La mise en oeuvre de best practices....883
40.15.3. L'utilisation des connexions et des Statements....883
40.15.4. L'utilisation d'un pool de connexions....883
40.15.5. La configuration et l'utilisation des ResultSets en fonction des besoins....883

Table des matières

40. JDBC (Java DataBase Connectivity)
40.15.6. L'utilisation des PreparedStatement....884
40.15.7. La maximisation des traitements effectués par la base de données :....884
40.15.8. L'exécution de plusieurs requêtes en mode batch....884
40.15.9. Prêter une attention particulière aux transactions....884
40.15.10. L'utilisation des fonctionnalités de JDBC 3.0....885
40.15.11. Les optimisations sur la base de données....885
40.15.12. L'utilisation d'un cache....885
40.16. Les ressources relatives à JDBC....886

41. JDO (Java Data Object)....887
41.1. La présentation de JDO....887
41.2. Un exemple avec Lido....888
41.2.1. La création de la classe qui va encapsuler les données....890
41.2.2. La création de l'objet qui va assurer les actions sur les données....890
41.2.3. La compilation....891
41.2.4. La définition d'un fichier metadata....891
41.2.5. L'enrichissement des classes contenant des données....892
41.2.6. La définition du schéma de la base de données....892
41.2.7. L'exécution de l'exemple....894
41.3. L'API JDO....894
41.3.1. L'interface PersistenceManager....895
41.3.2. L'interface PersistenceManagerFactory....895
41.3.3. L'interface PersistenceCapable....895
41.3.4. L'interface Query....896
41.3.5. L'interface Transaction....896
41.3.6. L'interface Extent....896
41.3.7. La classe JDOHelper....896
41.4. La mise en oeuvre....897
41.4.1. La définition d'une classe qui va encapsuler les données....897
41.4.2. La définition d'une classe qui va utiliser les données....898
41.4.3. La compilation des classes....898
41.4.4. La définition d'un fichier de description....898
41.4.5. L'enrichissement de la classe qui va contenir les données....898
41.5. Le parcours de toutes les occurrences....899
41.6. La mise en oeuvre de requêtes....900

42. Hibernate....902
42.1. La création d'une classe qui va encapsuler les données....903
42.2. La création d'un fichier de correspondance....904
42.3. Les propriétés de configuration....906
42.4. L'utilisation d'Hibernate....907
42.5. La persistance d'une nouvelle occurrence....908
42.6. L'obtention d'une occurrence à partir de son identifiant....910
42.7. Le langage de requête HQL....911
42.8. La mise à jour d'une occurrence....915
42.9. La suppression d'une ou plusieurs occurrences....915
42.10. Les relations....915
42.10.1. Les relations un / un....916
42.10.1.1. Le mapping avec un Component....916
42.10.1.2. Le mapping avec une relation One-to-One avec clé primaire partagée....924
42.10.1.3. Le mapping avec une relation One-to-One avec clé étrangère....933
42.11. Les outils de génération de code....941

43. JPA (Java Persistence API)....942
43.1. L'installation de l'implémentation de référence....942
43.2. Les entités....943
43.2.1. Le mapping entre le bean entité et la table....943
43.2.2. Le mapping de propriété complexe....953

Table des matières

43. JPA (Java Persistence API)
43.2.3. Le mapping d'une entité sur plusieurs tables....955
43.2.4. L'utilisation d'objets embarqués dans les entités....958
43.3. Le fichier de configuration du mapping....959
43.4. L'utilisation du bean entité....960
43.4.1. L'utilisation du bean entité....960
43.4.2. L'EntityManager....960
43.4.2.1. L'obtention d'une instance de la classe EntityManager....961
43.4.2.2. L'utilisation de la classe EntityManager....961
43.4.2.3. L'utilisation de la classe EntityManager pour la création d'une occurrence....962
43.4.2.4. L'utilisation de la classe EntityManager pour rechercher des occurrences....962
43.4.2.5. L'utilisation de la classe EntityManager pour rechercher des données par requête....963
43.4.2.6. L'utilisation de la classe EntityManager pour modifier une occurrence....965
43.4.2.7. L'utilisation de la classe EntityManager pour fusionner des données....966
43.4.2.8. L'utilisation de la classe EntityManager pour supprimer une occurrence....966
43.4.2.9. L'utilisation de la classe EntityManager pour rafraîchir les données d'une occurrence....967
43.5. Le fichier persistence.xml....968
43.6. La gestion des transactions hors Java EE....969
43.7. La gestion des relations entre tables dans le mapping....970
43.8. Les callbacks d'événements....970

Partie 6 : La machine virtuelle Java (JVM)....972

44. La JVM (Java Virtual Machine)....973
44.1. La mémoire de la JVM....974
44.1.1. Le Java Memory Model....974
44.1.2. Les différentes zones de la mémoire....974
44.1.2.1. La Pile (Stack)....975
44.1.2.2. Le tas (Heap)....975
44.1.2.3. La zone de mémoire "Method area"....975
44.1.2.4. La zone de mémoire "Code Cache"....975
44.2. Le cycle de vie d'une classe dans la JVM....975
44.2.1. Le chargement des classes....976
44.2.1.1. La recherche des fichiers .class....977
44.2.1.2. Le chargement du byte code....978
44.2.2. La liaison de la classe....978
44.2.3. L'initialisation de la classe....979
44.2.4. Le chargement des classes et la police de sécurité....980
44.3. Les ClassLoaders....980
44.3.1. Le mode de fonctionnement d'un ClassLoader....981
44.3.2. La délégation du chargement d'une classe....984
44.3.2.1. L'écriture d'un classloader personnalisé....985
44.4. Le bytecode....986
44.4.1. L'outil Jclasslib bytecode viewer....987
44.4.2. Le jeu d'instructions de la JVM....988
44.4.3. Le format des fichiers .class....989
44.5. Le compilateur JIT....989
44.6. Les paramètres de la JVM HotSpot....990
44.7. Les interactions de la machine virtuelle avec des outils externes....995
44.7.1. L'API Java Virtual Machine Debug Interface (JVMDI)....995
44.7.2. L'API Java Virtual Machine Profiler Interface (JVMPI)....995
44.7.3. L'API Java Virtual Machine Tools Interface (JVMTI)....996
44.7.4. L'architecture Java Platform Debugger Architecture (JPDA)....996
44.7.5. Des outils de profiling....996

45. La gestion de la mémoire....997
45.1. Le ramasse miettes (Garbage Collector ou GC)....997
45.1.1. Le rôle du ramasse miettes....998
45.1.2. Les différents concepts des algorithmes du ramasse miettes....998

Table des matières

45. La gestion de la mémoire
45.1.3. L'utilisation de générations....999
45.1.4. Les limitations du ramasse miettes....1000
45.1.5. Les facteurs de performance du ramasse miettes....1001
45.2. Le fonctionnement du ramasse miettes de la JVM Hotspot....1001
45.2.1. Les trois générations utilisées....1002
45.2.2. Les algorithmes d'implémentation du GC....1004
45.2.2.1. Le Serial Collector....1004
45.2.2.2. Le Parallel collector ou throughput collector....1006
45.2.2.3. Le Parallel Compacting Collector....1008
45.2.2.4. Le Concurrent Mark-Sweep (CMS) Collector....1009
45.2.3. L'auto sélection des paramètres du GC par la JVM Hotspot....1013
45.2.4. La sélection explicite d'un algorithme pour le GC....1014
45.2.5. Demander l'exécution d'une collection majeure.....1015
45.2.6. La méthode finalize()....1015
45.3. Le paramétrage du ramasse miettes de la JVM HotSpot....1016
45.3.1. Les options pour configurer le ramasse miettes....1016
45.3.2. La configuration de la JVM pour améliorer les performances du GC....1018
45.4. Le monitoring de l'activité du ramasse miettes....1020
45.5. Les différents types de référence....1020
45.6. L'obtention d'informations sur la mémoire de la JVM....1021
45.7. Les fuites de mémoire (Memory leak)....1021
45.7.1. Diagnostiquer une fuite de mémoire....1022
45.8. Les exceptions liées à un manque de mémoire....1023
45.8.1. L'exception de type StackOverFlowError....1023
45.8.2. L'exception de type OutOfMemoryError....1023
45.8.2.1. L'exception OutOfMemoryError : Java heap space....1024
45.8.2.2. L'exception OutOfMemoryError : PermGen space....1024
45.8.2.3. L'exception OutOfMemoryError : Requested array size exceeds VM limit....1024

46. La décompilation et l'obfuscation....1025
46.1. Décompiler du byte code....1025
46.1.1. JAD : the fast Java Decompiler....1025
46.1.2. La mise en oeuvre et les limites de la décompilation....1026
46.2. Obfusquer le byte code....1030
46.2.1. Le mode de fonctionnement de l'obfuscation....1031
46.2.2. Un exemple de mise en oeuvre avec ProGuard....1032
46.2.3. Les problèmes possibles lors de l'obfuscation....1036
46.2.4. L'utilisation d'un ClassLoader dédié....1036

47. Terracotta....1037
47.1. Présentation de Terrocatta....1038
47.1.1. Le mode de fonctionnement....1039
47.1.2. La gestion des objets par le cluster....1040
47.1.3. Les avantages de Terracotta....1041
47.1.4. L'architecture d'un cluster Terracotta....1041
47.2. Les concepts utilisés....1042
47.2.1. Les racines (root)....1042
47.2.2. Les transactions....1043
47.2.3. Les verrous (lock)....1043
47.2.4. L'instrumentation des classes....1045
47.2.5. L'invocation distribuée de méthodes....1046
47.2.6. Le ramasse miette distribué....1046
47.3. La mise en oeuvre des fonctionnalités....1046
47.3.1. L'installation....1046
47.3.2. Les modules d'intégration....1047
47.3.3. Les scripts de commande....1048
47.3.4. Les limitations....1048
47.4. Les cas d'utilisation....1049

Table des matières

47. Terracotta
47.4.1. La réplication de sessions HTTP....1049
47.4.2. Un cache distribué....1050
47.4.3. La répartition de charge de traitements....1050
47.4.4. Le partage de données entre applications....1050
47.5. Quelques exemples de mise en oeuvre....1051
47.5.1. Un exemple simple avec une application standalone....1051
47.5.2. Un second exemple avec une application standalone....1053
47.5.3. Un exemple avec une application web....1055
47.5.4. La réplication de sessions sous Tomcat....1057
47.5.5. Le partage de données entre deux applications....1057
47.6. La console développeur....1060
47.7. Le fichier de configuration....1060
47.7.1. La configuration de la partie system....1061
47.7.2. La configuration de la partie serveur....1061
47.7.3. La configuration de la partie cliente....1063
47.7.4. La configuration de la partie applicative....1063
47.7.4.1. La définition des racines....1063
47.7.4.2. La définition des verrous....1064
47.7.4.3. La définition des classes à instrumenter....1065
47.7.4.4. La définition des champs qui ne doivent pas être gérés....1065
47.7.4.5. La définition des méthodes dont l'invocation doit être distribuée....1066
47.7.4.6. La définition des webapps dont la session doit être gérée....1066
47.8. La fiabilisation du cluster....1066
47.8.1. La configuration minimale....1067
47.8.2. La configuration pour la fiabilité....1068
47.8.3. La configuration pour une haute disponibilité....1069
47.8.4. La configuration pour une haute disponibilité et la fiabilité....1070
47.8.5. La configuration pour un environnement de production....1072
47.8.6. La configuration pour la montée en charge....1072
47.9. Quelques recommandations....1072

Partie 7 : Développement d'applications d'entreprises....1074

48. J2EE / Java EE....1075
48.1. La présentation de J2EE....1075
48.2. Les API de J2EE....1076
48.3. L'environnement d'exécution des applications J2EE....1077
48.3.1. Les conteneurs....1077
48.3.2. Le conteneur web....1078
48.3.3. Le conteneur d'EJB....1078
48.3.4. Les services proposés par la plate-forme J2EE....1078
48.4. L'assemblage et le déploiement d'applications J2EE....1079
48.4.1. Le contenu et l'organisation d'un fichier EAR....1079
48.4.2. La création d'un fichier EAR....1079
48.4.3. Les limitations des fichiers EAR....1079
48.5. J2EE 1.4 SDK....1079
48.5.1. L'installation de l'implémentation de référence sous Windows....1080
48.5.2. Le démarrage et l'arrêt du serveur....1081
48.5.3. L'outil asadmin....1083
48.5.4. Le déploiement d'applications....1083
48.5.5. La console d'administration....1084
48.6. La présentation de Java EE 5.0....1084
48.6.1. La simplification des développements....1085
48.6.2. La version 3.0 des EJB....1086
48.6.3. Un accès facilité aux ressources grâce à l'injection de dépendance....1087
48.6.4. JPA : le nouvelle API de persistance....1087
48.6.5. Des services web plus simple à écrire....1089
48.6.6. Le développement d'applications Web....1090

Table des matières

48. J2EE / Java EE
48.6.7. Les autres fonctionnalités....1090
48.6.8. L'installation du SDK Java EE 5 sous Windows....1090
48.7. La présentation de Java EE 6....1092
48.7.1. Les spécifications de Java EE 6....1092
48.7.2. Une plate-forme plus légère....1093
48.7.2.1. La notion de profile....1094
48.7.2.2. La notion de pruning....1094
48.7.3. Les évolutions dans les spécifications existantes....1095
48.7.3.1. Servlet 3.0....1095
48.7.3.2. JSF 2.0....1095
48.7.3.3. Les EJB 3.1....1096
48.7.3.4. JPA 2.0....1096
48.7.3.5. JAX-WS 2.2....1096
48.7.3.6. Interceptors 1.1....1097
48.7.4. Les nouvelles API....1097
48.7.4.1. JAX-RS 1.1....1097
48.7.4.2. Contexte and Dependency Injection (WebBeans) 1.0....1097
48.7.4.3. Dependency Injection 1.0....1098
48.7.4.4. Bean Validation 1.0....1098
48.7.4.5. Managed Beans 1.0....1098

49. JavaMail....1099
49.1. Le téléchargement et l'installation....1099
49.2. Les principaux protocoles....1100
49.2.1. Le protocole SMTP....1100
49.2.2. Le protocole POP....1100
49.2.3. Le protocole IMAP....1100
49.2.4. Le protocole NNTP....1100
49.3. Les principales classes et interfaces de l'API JavaMail....1100
49.3.1. La classe Session....1101
49.3.2. Les classes Address, InternetAddress et NewsAddress....1101
49.3.3. L'interface Part....1102
49.3.4. La classe Message....1102
49.3.5. Les classes Flags et Flag....1104
49.3.6. La classe Transport....1105
49.3.7. La classe Store....1105
49.3.8. La classe Folder....1105
49.3.9. Les propriétés d'environnement....1106
49.3.10. La classe Authenticator....1106
49.4. L'envoi d'un e mail par SMTP....1107
49.5. La récupération des messages d'un serveur POP3....1108
49.6. Les fichiers de configuration....1108
49.6.1. Les fichiers javamail.providers et javamail.default.providers....1108
49.6.2. Les fichiers javamail.address.map et javamail.default.address.map....1109

50. JMS (Java Messaging Service)....1110
50.1. La présentation de JMS....1110
50.2. Les services de messages....1111
50.3. Le package javax.jms....1112
50.3.1. La fabrique de connexion....1112
50.3.2. L'interface Connection....1112
50.3.3. L'interface Session....1113
50.3.4. Les messages....1114
50.3.4.1. L'en tête....1114
50.3.4.2. Les propriétés....1115
50.3.4.3. Le corps du message....1115
50.3.5. L'envoi de messages....1115
50.3.6. La réception de messages....1116

Table des matières

50. JMS (Java Messaging Service)
50.4. L'utilisation du mode point à point (queue)....1116
50.4.1. La création d'une factory de connexion : QueueConnectionFactory....1116
50.4.2. L'interface QueueConnection....1117
50.4.3. La session : l'interface QueueSession....1117
50.4.4. L'interface Queue....1117
50.4.5. La création d'un message....1118
50.4.6. L'envoi de messages : l'interface QueueSender....1118
50.4.7. La réception de messages : l'interface QueueReceiver....1118
50.4.7.1. La réception dans le mode synchrone....1119
50.4.7.2. La réception dans le mode asynchrone....1119
50.4.7.3. La sélection de messages....1119
50.5. L'utilisation du mode publication/abonnement (publish/subscribe)....1120
50.5.1. La création d'une factory de connexion : TopicConnectionFactory....1120
50.5.2. L'interface TopicConnection....1120
50.5.3. La session : l'interface TopicSession....1120
50.5.4. L'interface Topic....1121
50.5.5. La création d'un message....1121
50.5.6. L'émission de messages : l'interface TopicPublisher....1121
50.5.7. La réception de messages : l'interface TopicSubscriber....1121
50.6. La gestion des erreurs....1122
50.6.1. Les exceptions de JMS....1122
50.6.2. L'interface ExceptionListener....1122
50.7. JMS 1.1....1122
50.7.1. L'utilisation de l'API JMS 1.0 et 1.1....1123
50.7.2. L'interface ConnectionFactory....1124
50.7.3. L'interface Connection....1124
50.7.4. L'interface Session....1125
50.7.5. L'interface Destination....1126
50.7.6. L'interface MessageProducer....1126
50.7.7. L'interface MessageConsumer....1127
50.7.7.1. La réception synchrone de messages....1127
50.7.7.2. La réception asynchrone de messages....1127
50.7.8. Le filtrage des messages....1128
50.7.8.1. La définition du filtre....1128
50.7.9. Des exemples de mise en oeuvre....1129
50.8. Les ressources relatives à JMS....1131

51. Les EJB (Entreprise Java Bean)....1132
51.1. La présentation des EJB....1133
51.1.1. Les différents types d'EJB....1134
51.1.2. Le développement d'un EJB....1134
51.1.3. L'interface remote....1135
51.1.4. L'interface home....1135
51.2. Les EJB session....1136
51.2.1. Les EJB session sans état....1137
51.2.2. Les EJB session avec état....1138
51.3. Les EJB entité....1138
51.4. Les outils pour développer et mettre oeuvre des EJB....1139
51.4.1. Les outils de développement....1139
51.4.2. Les conteneurs d'EJB....1139
51.4.2.1. JBoss....1139
51.5. Le déploiement des EJB....1139
51.5.1. Le descripteur de déploiement....1140
51.5.2. Le mise en package des beans....1140
51.6. L'appel d'un EJB par un client....1140
51.6.1. Un exemple d'appel d'un EJB session....1140
51.7. Les EJB orientés messages....1141

Table des matières

52. Les EJB 3....1142
52.1. L'historique des EJB....1142
52.2. Les nouveaux concepts et fonctionnalités utilisés....1143
52.2.1. L'utilisation de POJO et POJI....1143
52.2.2. L'utilisation des annotations....1144
52.2.3. L'injection de dépendances....1145
52.2.4. La configuration par défaut....1146
52.2.5. Les intercepteurs....1146
52.3. EJB 2.x vs EJB 3.0....1146
52.4. Les conventions de nommage....1147
52.5. Les EJB de type Session....1147
52.5.1. L'interface distante et/ou locale....1147
52.5.2. Les beans de type Stateless....1148
52.5.3. Les beans de type stateful....1149
52.5.4. L'invocation d'un EJB Session via un service web....1150
52.5.5. L'utilisation des exceptions....1150
52.6. Les EJB de type Entity....1151
52.6.1. La création d'un bean Entity....1151
52.6.2. La persistance des entités....1153
52.6.3. La création d'un EJB Session pour manipuler le bean Entity....1154
52.7. Un exemple simple complet....1154
52.7.1. La création de l'entité....1155
52.7.2. La création de la façade....1156
52.7.3. La création du service web....1157
52.8. L'utilisation des EJB par un client....1158
52.8.1. Pour un client de type application standalone....1159
52.8.2. Pour un client de type module Application Client Java EE....1159
52.9. L'injection de dépendances....1159
52.9.1. L'annotation @javax.ejb.EJB....1160
52.9.2. L'annotation @javax.annotation.Resource....1160
52.9.3. Les annotations @javax.annotation.Resources et @javax.ejb.EJBs....1161
52.9.4. L'annotation @javax.xml.ws.WebServiceRef....1161
52.10. Les intercepteurs....1161
52.10.1. Le développement d'un intercepteur....1162
52.10.1.1. L'interface InvocationContext....1162
52.10.1.2. La définition d'un intercepteur lié aux méthodes métier....1163
52.10.1.3. La définition d'un intercepteur lié au cycle de vie....1163
52.10.1.4. La mise oeuvre d'une classe d'un intercepteur....1164
52.10.2. Les intercepteurs par défaut....1164
52.10.3. Les annotations des intercepteurs....1165
52.10.3.1. L'annotation @javax.annotation.PostContruct....1165
52.10.3.2. L'annotation @javax.annotation.PreDestroy....1166
52.10.3.3. L'annotation @javax.interceptor.AroundInvoke....1166
52.10.3.4. L'annotation @javax.interceptor.ExcludeClassInterceptors....1166
52.10.3.5. L'annotation @javax.interceptor.ExcludeDefaultInterceptors....1166
52.10.3.6. L'annotation @javax.interceptor.Interceptors....1167
52.10.4. L'utilisation d'un intercepteur....1167
52.11. Les EJB de type MessageDriven....1168
52.11.1. L'annotation @ javax.ejb.MessageDriven....1169
52.11.2. L'annotation @javax.ejb.ActivationConfigProperty....1169
52.11.3. Un exemple d'EJB de type MDB....1169
52.12. Le packaging des EJB....1173
52.13. Les tests des EJB....1173
52.14. Les transactions....1174
52.14.1. La mise en oeuvre des transactions dans les EJB....1174
52.14.2. La définition de transactions....1175
52.14.2.1. La définition du mode de gestion des transactions dans un EJB....1175
52.14.2.2. La définition de transactions avec l'annotation @TransactionAttribute....1175
52.14.2.3. La définition de transactions dans le descripteur de déploiement....1176

Table des matières

52. Les EJB 3
52.14.2.4. Des recommandations sur la mise en oeuvre des transactions....1176
52.15. La mise en oeuvre de la sécurité....1177
52.15.1. L'authentification et l'identification de l'utilisateur....1177
52.15.2. La définition des restrictions....1177
52.15.2.1. La définition des restrictions avec les annotations....1177
52.15.2.2. La définition des restrictions avec le descripteur de déploiement....1178
52.15.3. Les annotations pour la sécurité....1178
52.15.3.1. javax.annotation.security.DeclareRoles....1178
52.15.3.2. javax.annotation.security.DenyAll....1179
52.15.3.3. javax.annotation.security.PermitAll....1179
52.15.3.4. javax.annotation.security.RolesAllowed....1179
52.15.3.5. javax.annotation.security.RunAs....1179
52.15.4. La mise oeuvre de la sécurité par programmation....1180

53. Les EJB 3.1....1181
53.1. Les interfaces locales sont optionnelles....1181
53.2. Les EJB Singleton....1183
53.3. EJB Lite....1186
53.4. La simplification du packaging....1190
53.5. Les améliorations du service Timer....1193
53.5.1. La définition d'un timer....1193
53.5.2. L'annotation @Schedule....1195
53.5.3. La persistance des timers....1197
53.5.4. L'interface Timer....1198
53.5.5. L'interface TimerService....1199
53.6. La standardisation des noms JNDI....1201
53.7. L'invocation asynchrone des EJB session....1202
53.7.1. L'annotation @Asynchronous....1202
53.7.2. L'invocation d'une méthode asynchrone....1204
53.8. L'invocation d'un EJB hors du conteneur....1207

54. Les services web de type Soap....1209
54.1. La présentation des services web....1210
54.1.1. La définition d'un service web....1210
54.1.2. Les différentes utilisations....1211
54.2. Les standards....1212
54.2.1. SOAP....1212
54.2.1.1. La structure des messages SOAP....1213
54.2.1.2. L'encodage des messages SOAP....1214
54.2.1.3. Les différentes versions de SOAP....1214
54.2.2. WSDL....1215
54.2.2.1. Le format général d'un WSDL....1215
54.2.2.2. L'élément Types....1218
54.2.2.3. L'élément Message....1218
54.2.2.4. L'élément PortType/Interface....1218
54.2.2.5. L'élément Binding....1219
54.2.2.6. L'élément Service....1220
54.2.3. Les registres et les services de recherche....1221
54.2.3.1. UDDI....1221
54.2.3.2. Ebxml....1221
54.3. Les différents formats de services web SOAP....1222
54.3.1. Le format RPC Encoding....1224
54.3.2. Le format RPC Literal....1226
54.3.3. Le format Document encoding....1228
54.3.4. Le format Document literal....1228
54.3.5. Le format Document Literal wrapped....1231
54.3.6. Le choix du format à utiliser....1233
54.3.6.1. L'utilisation de document/literal....1234

Table des matières

54. Les services web de type Soap
54.3.6.2. L'utilisation de document/literal wrapped....1234
54.3.6.3. L'utilisation de RPC/Literal....1234
54.3.6.4. L'utilisation de RPC/encoded....1235
54.4. Des conseils pour la mise en oeuvre....1235
54.4.1. Les étapes de la mise en oeuvre....1235
54.4.2. Quelques recommandations....1236
54.4.3. Les problèmes liés à SOAP....1237
54.5. Les API Java pour les services web....1237
54.5.1. JAX-RPC....1238
54.5.1.1. La mise en oeuvre côté serveur....1239
54.5.1.2. La mise en oeuvre côté client....1240
54.5.2. JAXM....1240
54.5.3. JAXR....1241
54.5.4. SAAJ....1241
54.5.5. JAX-WS....1241
54.5.5.1. La mise en oeuvre de JAX-WS....1242
54.5.5.2. La production de service web avec JAX-WS....1242
54.5.5.3. La consommation de services web avec JAX-WS....1246
54.5.5.4. Les handlers....1247
54.5.6. La JSR 181 (Web Services Metadata for the Java Platform)....1247
54.5.6.1. Les annotations définies....1248
54.5.6.2. javax.jws.WebService....1248
54.5.6.3. javax.jws.WebMethod....1249
54.5.6.4. javax.jws.OneWay....1249
54.5.6.5. javax.jws.WebParam....1250
54.5.6.6. javax.jws.WebResult....1250
54.5.6.7. javax.jws.soap.SOAPBinding....1251
54.5.6.8. javax.jws.HandlerChain....1252
54.5.6.9. javax.jws.soap.SOAPMessageHandlers....1252
54.5.7. La JSR 224 (JAX-WS 2.0 Annotations)....1252
54.5.7.1. javax.xml.ws.BindingType....1252
54.5.7.2. javax.xml.ws.RequestWrapper....1252
54.5.7.3. javax.xml.ws.ResponseWrapper....1253
54.5.7.4. javax.xml.ws.ServiceMode....1253
54.5.7.5. javax.xml.ws.WebFault....1254
54.5.7.6. javax.xml.ws.WebEndpoint....1254
54.5.7.7. javax.xml.ws.WebServiceclient....1254
54.5.7.8. javax.xml.ws.WebServiceProvider....1254
54.5.7.9. javax.xml.ws.WebServiceRef....1255
54.6. Les implémentations des services web....1255
54.6.1. Axis 1.0....1256
54.6.1.1. Installation....1257
54.6.1.2. La mise en oeuvre côté serveur....1259
54.6.1.3. Mise en oeuvre côté client....1260
54.6.1.4. L'outil TCPMonitor....1262
54.6.2. Apache Axis 2....1262
54.6.3. Xfire....1263
54.6.4. Apache CXF....1263
54.6.5. JWSDP (Java Web Service Developer Pack)....1263
54.6.5.1. L'installation du JWSDP 1.1....1264
54.6.5.2. L'exécution du serveur....1264
54.6.5.3. L'exécution d'un des exemples....1265
54.6.6. Java EE 5....1267
54.6.7. Java SE 6....1267
54.6.8. Le projet Metro et WSIT....1270
54.7. Inclure des pièces jointes dans SOAP....1270
54.8. WS-I....1270
54.8.1. WS-I Basic Profile....1271

Table des matières

54. Les services web de type Soap
54.9. Les autres spécifications....1271

Partie 8 : Le développement d'applications web....1273

55. Les servlets....1274
55.1. La présentation des servlets....1274
55.1.1. Le fonctionnement d'une servlet (cas d'utilisation de http)....1275
55.1.2. Les outils nécessaires pour développer des servlets....1275
55.1.3. Le rôle du conteneur web....1276
55.1.4. Les différences entre les servlets et les CGI....1276
55.2. L'API servlet....1276
55.2.1. L'interface Servlet....1277
55.2.2. La requête et la réponse....1278
55.2.3. Un exemple de servlet....1278
55.3. Le protocole HTTP....1279
55.4. Les servlets http....1280
55.4.1. La méthode init()....1281
55.4.2. L'analyse de la requête....1281
55.4.3. La méthode doGet()....1281
55.4.4. La méthode doPost()....1282
55.4.5. La génération de la réponse....1282
55.4.6. Un exemple de servlet HTTP très simple....1284
55.5. Les informations sur l'environnement d'exécution des servlets....1285
55.5.1. Les paramètres d'initialisation....1285
55.5.2. L'objet ServletContext....1286
55.5.3. Les informations contenues dans la requête....1287
55.6. L'utilisation des cookies....1289
55.6.1. La classe Cookie....1289
55.6.2. L'enregistrement et la lecture d'un cookie....1289
55.7. Le partage d'informations entre plusieurs échanges HTTP....1290
55.8. Packager une application web....1290
55.8.1. La structure d'un fichier .war....1290
55.8.2. Le fichier web.xml....1291
55.8.3. Le déploiement d'une application web....1293
55.9. L'utilisation Log4J dans une servlet....1293

56. Les JSP (Java Server Pages)....1297
56.1. La présentation des JSP....1297
56.1.1. Le choix entre JSP et Servlets....1298
56.1.2. Les JSP et les technologies concurrentes....1298
56.2. Les outils nécessaires....1299
56.2.1. L'outil JavaServer Web Development Kit (JSWDK) sous Windows....1299
56.2.2. Le serveur Tomcat....1301
56.3. Le code HTML....1301
56.4. Les Tags JSP....1301
56.4.1. Les tags de directives <%@ ... %>....1301
56.4.1.1. La directive page....1301
56.4.1.2. La directive include....1303
56.4.1.3. La directive taglib....1304
56.4.2. Les tags de scripting....1304
56.4.2.1. Le tag de déclarations <%! ... %>....1304
56.4.2.2. Le tag d'expressions <%= ... %>....1305
56.4.2.3. Les variables implicites....1306
56.4.2.4. Le tag des scriptlets <% ... %>....1306
56.4.3. Les tags de commentaires....1307
56.4.3.1. Les commentaires HTML <!-- ... -->....1307
56.4.3.2. Les commentaires cachés <%-- ... --%>....1308
56.4.4. Les tags d'actions....1308

Table des matières

56. Les JSP (Java Server Pages)
56.4.4.1. Le tag <jsp:useBean> 1308
56.4.4.2. Le tag <jsp:setProperty > 1311
56.4.4.3. Le tag <jsp:getProperty> 1312
56.4.4.4. Le tag de redirection <jsp:forward> 1313
56.4.4.5. Le tag <jsp:include> 1314
56.4.4.6. Le tag <jsp:plugin> 1314
56.5. Un Exemple très simple 1315
56.6. La gestion des erreurs 1316
56.6.1. La définition d'une page d'erreur 1316
56.7. Les bibliothèques de tags personnalisés (custom taglibs) 1316
56.7.1. La présentation des tags personnalisés 1317
56.7.2. Les handlers de tags 1317
56.7.3. L'interface Tag 1318
56.7.4. L'accès aux variables implicites de la JSP 1319
56.7.5. Les deux types de handlers 1319
56.7.5.1. Les handlers de tags sans corps 1319
56.7.5.2. Les handlers de tags avec corps 1320
56.7.6. Les paramètres d'un tag 1320
56.7.7. La définition du fichier de description de la bibliothèque de tags (TLD) 1320
56.7.8. L'utilisation d'une bibliothèque de tags 1322
56.7.8.1. L'utilisation dans le code source d'une JSP 1323
56.7.8.2. Le déploiement d'une bibliothèque 1324
56.7.9. Le déploiement et les tests dans Tomcat 1324
56.7.10. Les bibliothèques de tags existantes 1325
56.7.10.1. Struts 1325
56.7.10.2. Jakarta Tag libs 1325
56.7.10.3. JSP Standard Tag Library (JSTL) 1325

57. JSTL (Java server page Standard Tag Library) 1327
57.1. Un exemple simple 1328
57.2. Le langage EL (Expression Langage) 1329
57.3. La bibliothèque Core 1331
57.3.1. Le tag set 1331
57.3.2. Le tag out 1332
57.3.3. Le tag remove 1333
57.3.4. Le tag catch 1333
57.3.5. Le tag if 1334
57.3.6. Le tag choose 1335
57.3.7. Le tag forEach 1335
57.3.8. Le tag forTokens 1337
57.3.9. Le tag import 1338
57.3.10. Le tag redirect 1339
57.3.11. Le tag url 1339
57.4. La bibliothèque XML 1340
57.4.1. Le tag parse 1341
57.4.2. Le tag set 1341
57.4.3. Le tag out 1342
57.4.4. Le tag if 1342
57.4.5. Le tag choose 1343
57.4.6. Le tag forEach 1343
57.4.7. Le tag transform 1343
57.5. La bibliothèque I18n 1344
57.5.1. Le tag bundle 1345
57.5.2. Le tag setBundle 1345
57.5.3. Le tag message 1346
57.5.4. Le tag setLocale 1346
57.5.5. Le tag formatNumber 1347
57.5.6. Le tag parseNumber 1347

Table des matières

57. JSTL (Java server page Standard Tag Library)
57.5.7. Le tag formatDate....1348
57.5.8. Le tag parseDate....1348
57.5.9. Le tag setTimeZone....1349
57.5.10. Le tag timeZone....1349
57.6. La bibliothèque Database....1349
57.6.1. Le tag setDataSource....1350
57.6.2. Le tag query....1350
57.6.3. Le tag transaction....1352
57.6.4. Le tag update....1352

58. Struts....1354
58.1. L'installation et la mise en oeuvre....1355
58.1.1. Un exemple très simple....1357
58.2. Le développement des vues....1361
58.2.1. Les objets de type ActionForm....1362
58.2.2. Les objets de type DynaActionForm....1363
58.3. La configuration de Struts....1364
58.3.1. Le fichier struts-config.xml....1364
58.3.2. La classe ActionMapping....1366
58.3.3. Le développement de la partie contrôleur....1366
58.3.4. La servlet de type ActionServlet....1367
58.3.5. La classe Action....1368
58.3.6. La classe DispatchAction....1370
58.3.7. La classe LookupDispatchAction....1373
58.3.8. La classe ForwardAction....1375
58.4. Les bibliothèques de tags personnalisés....1376
58.4.1. La bibliothèque de tags HTML....1376
58.4.1.1. Le tag <html:html>....1378
58.4.1.2. Le tag <html:form>....1378
58.4.1.3. Le tag <html:button>....1378
58.4.1.4. Le tag <html:cancel>....1379
58.4.1.5. Le tag <html:submit>....1379
58.4.1.6. Le tag <html:radio>....1379
58.4.1.7. Le tag <html:checkbox>....1380
58.4.2. La bibliothèque de tags Bean....1380
58.4.2.1. Le tag <bean:cookie>....1380
58.4.2.2. Le tag <bean:define>....1381
58.4.2.3. Le tag <bean:header>....1381
58.4.2.4. Le tag <bean:include>....1381
58.4.2.5. Le tag <bean:message>....1382
58.4.2.6. Le tag <bean:page>....1382
58.4.2.7. Le tag <bean:param>....1382
58.4.2.8. Le tag <bean:resource>....1382
58.4.2.9. Le tag <bean:size>....1383
58.4.2.10. Le tag <bean:struts>....1383
58.4.2.11. Le tag <bean:write>....1383
58.4.3. La bibliothèque de tags Logic....1384
58.4.3.1. Les tags <logic:empty> et <logic:notEmpty>....1386
58.4.3.2. Les tags <logic:equal> et <logic:notEqual>....1386
58.4.3.3. Les tags <logic:lessEqual>, <logic:lessThan>, <logic:greaterEqual>, et <logic:greaterThan>....1387
58.4.3.4. Les tags <logic:match> et <logic:notMatch>....1387
58.4.3.5. Les tags <logic:present> et <logic:notPresent>....1387
58.4.3.6. Le tag <logic:forward>....1388
58.4.3.7. Le tag <logic:redirect>....1388
58.4.3.8. Le tag <logic:iterate>....1389
58.5. La validation de données....1389
58.5.1. La classe ActionError....1389

Table des matières

58. Struts
58.5.2. La classe ActionErrors....1390
58.5.3. L'affichage des messages d'erreur....1391
58.5.4. Les classes ActionMessage et ActionMessages....1394
58.5.5. L'affichage des messages....1394

59. JSF (Java Server Faces)....1396
59.1. La présentationde JSF....1396
59.2. Le cycle de vie d'une requête....1397
59.3. Les implémentations....1398
59.3.1. L'implémentation de référence....1398
59.3.2. MyFaces....1399
59.4. Le contenu d'une application....1399
59.5. La configuration de l'application....1401
59.5.1. Le fichier web.xml....1401
59.5.2. Le fichier faces-config.xml....1402
59.6. Les beans....1404
59.6.1. Les beans managés (managed bean)....1404
59.6.2. Les expressions de liaison de données d'un bean....1405
59.6.3. Les Backing beans....1407
59.7. Les composants pour les interfaces graphiques....1408
59.7.1. Le modèle de rendu des composants....1409
59.7.2. L'utilisation de JSF dans une JSP....1409
59.8. La bibliothèque de tags Core....1410
59.8.1. Le tag <selectItem>....1410
59.8.2. Le tag <selectItems>....1411
59.8.3. Le tag <verbatim>....1412
59.8.4. Le tag <attribute>....1413
59.8.5. Le tag <facet>....1413
59.9. La bibliothèque de tags Html....1413
59.9.1. Les attributs communs....1414
59.9.2. Le tag <form>....1417
59.9.3. Les tags <inputText>, <inputTextarea>, <inputSecret>....1417
59.9.4. Le tag <ouputText> et <outputFormat>....1418
59.9.5. Le tag <graphicImage>....1420
59.9.6. Le tag <inputHidden>....1420
59.9.7. Le tag <commandButton> et <commandLink>....1421
59.9.8. Le tag <ouputLink>....1422
59.9.9. Les tags <selectBooleanCheckbox> et <selectManyCheckbox>....1423
59.9.10. Le tag <selectOneRadio>....1425
59.9.11. Le tag <selectOneListbox>....1427
59.9.12. Le tag <selectManyListbox>....1427
59.9.13. Le tag <selectOneMenu>....1429
59.9.14. Le tag <selectManyMenu>....1430
59.9.15. Les tags <message> et <messages>....1430
59.9.16. Le tag <panelGroup>....1431
59.9.17. Le tag <panelGrid>....1432
59.9.18. Le tag <dataTable>....1433
59.10. La gestion et le stockage des données....1438
59.11. La conversion des données....1439
59.11.1. Le tag <convertNumber>....1439
59.11.2. Le tag <convertDateTime>....1441
59.11.3. L'affichage des erreurs de conversions....1442
59.11.4. L'écriture de convertisseurs personnalisés....1442
59.12. La validation des données....1443
59.12.1. Les classes de validation standard....1443
59.12.2. Contourner la validation....1444
59.12.3. L'écriture de classes de validation personnalisées....1444
59.12.4. La validation à l'aide de bean....1447

59. JSF (Java Server Faces)
59.12.5. La validation entre plusieurs composants....1448
59.12.6. L'écriture de tags pour un convertisseur ou un valideur de données....1450
59.12.6.1. L'écriture d'un tag personnalisé pour un convertisseur....1451
59.12.6.2. L'écriture d'un tag personnalisé pour un valideur....1451
59.13. La sauvegarde et la restauration de l'état....1451
59.14. Le système de navigation....1452
59.15. La gestion des événements....1453
59.15.1. Les événements liés à des changements de valeur....1454
59.15.2. Les événements liés à des actions....1456
59.15.3. L'attribut immediate....1457
59.15.4. Les événements liés au cycle de vie....1458
59.16. Le déploiement d'une application....1459
59.17. Un exemple d'application simple....1460
59.18. L'internationalisation....1463
59.19. Les points faibles de JSF....1467

60. D'autres frameworks pour les applications web....1469
60.1. stxx....1469
60.2. WebMacro....1469
60.3. FreeMarker....1469
60.4. Velocity....1470

Partie 9 : Le développement d'applications RIA / RDA....1471

61. Les applications riches : RIA et RDA....1472
61.1. Les applications de type RIA....1473
61.2. Les applications de type RDA....1473
61.3. Les contraintes....1473
61.4. Les solutions RIA....1474
61.4.1. Les solutions RIA reposant sur Java....1474
61.4.1.1. Sun Java FX....1474
61.4.1.2. Google GWT....1475
61.4.1.3. ZK....1475
61.4.1.4. Echo....1475
61.4.1.5. Apache Wicket....1476
61.4.1.6. Les composants JSF....1476
61.4.1.7. Tibco General Interface....1476
61.4.1.8. Eclipse RAP....1476
61.4.2. Les autres solutions RIA....1476
61.4.2.1. Adobe Flex....1477
61.4.2.2. Microsoft Silverlight....1477
61.4.2.3. Google Gears....1477
61.4.3. Une comparaison entre GWT et Flex....1477
61.5. Les solutions RDA....1478
61.5.1. Adobe AIR....1478
61.5.2. Eclipse RCP....1478
61.5.3. Netbeans RCP....1479

62. Les applets....1480
62.1. L'intégration d'applets dans une page HTML....1480
62.2. Les méthodes des applets....1481
62.2.1. La méthode init()....1481
62.2.2. La méthode start()....1481
62.2.3. La méthode stop()....1481
62.2.4. La méthode destroy()....1481
62.2.5. La méthode update()....1481
62.2.6. La méthode paint()....1482
62.2.7. Les méthodes size() et getSize()....1482

Table des matières

62. Les applets
62.2.8. Les méthodes getCodeBase() et getDocumentBase()....1483
62.2.9. La méthode showStatus()....1483
62.2.10. La méthode getAppletInfo()....1483
62.2.11. La méthode getParameterInfo()....1484
62.2.12. La méthode getGraphics()....1484
62.2.13. La méthode getAppletContext()....1484
62.2.14. La méthode setStub()....1484
62.3. Les interfaces utiles pour les applets....1484
62.3.1. L'interface Runnable....1484
62.3.2. L'interface ActionListener....1485
62.3.3. L'interface MouseListener pour répondre à un clic de souris....1485
62.4. La transmission de paramètres à une applet....1485
62.5. Les applets et le multimédia....1486
62.5.1. L'insertion d'images....1486
62.5.2. L'utilisation des capacités audio....1487
62.5.3. L'animation d'un logo....1488
62.6. Un applet pouvant s'exécuter comme une application....1489
62.7. Les droits des applets....1490

63. Java Web Start (JWS)....1491
63.1. La création du package de l'application....1491
63.2. La signature d'un fichier jar....1492
63.3. Le fichier JNPL....1492
63.4. La configuration du serveur web....1494
63.5. Le fichier HTML....1494
63.6. Le test de l'application....1495
63.7. L'utilisation du gestionnaire d'applications....1497
63.7.1. Le lancement d'une application....1498
63.7.2. L'affichage de la console....1499
63.7.3. Consigner les traces d'exécution dans un fichier de log....1499
63.8. L'API de Java Web Start....1499

64. Ajax....1500
64.1. La présentation d'Ajax....1501
64.2. Le détail du mode de fonctionnement....1504
64.3. Un exemple simple....1506
64.3.1. L'application de tests....1506
64.3.2. La prise en compte de l'événement déclencheur....1508
64.3.3. La création d'un objet de type XMLHttpRequest pour appeler la servlet....1508
64.3.4. L'exécution des traitements et le renvoie de la réponse par la servlet....1510
64.3.5. L'exploitation de la réponse....1511
64.3.6. L'exécution de l'application....1512
64.4. Des frameworks pour mettre en oeuvre Ajax....1513
64.4.1. Direct Web Remoting (DWR)....1513
64.4.1.1. Un exemple de mise en oeuvre de DWR....1514
64.4.1.2. Le fichier DWR.xml....1517
64.4.1.3. Les scripts engine.js et util.js....1520
64.4.1.4. Les scripts client générés....1521
64.4.1.5. Un exemple pour obtenir le contenu d'une page....1522
64.4.1.6. Un exemple pour valider des données....1523
64.4.1.7. Un exemple pour remplir dynamiquement une liste déroulante....1524
64.4.1.8. Un exemple pour afficher dynamiquement des informations....1526
64.4.1.9. Un exemple pour mettre à jour des données....1528
64.4.1.10. Un exemple pour remplir dynamiquement un tableau de données....1530
64.4.2. D'autres frameworks....1533

Table des matières

65. GWT (Google Web Toolkit)....1534
65.1. La présentation de GWT....1535
65.1.1. L'installation de GWT....1537
65.1.2. GWT version 1.6....1537
65.1.2.1. La nouvelle structure pour les projets....1537
65.1.2.2. Un nouveau système de gestion des événements....1538
65.1.2.3. De nouveaux composants....1539
65.1.3. GWT version 1.7....1540
65.2. La création d'une application....1540
65.2.1. L'application générée....1542
65.2.1.1. Le fichier MonApp.html....1542
65.2.1.2. Le fichier MonApp.gwt.xml....1543
65.2.1.3. Le fichier MonApp.java....1543
65.3. Les modes d'exécution....1544
65.3.1. Le mode hote (hosted mode)....1544
65.3.2. Le mode web (web mode)....1545
65.4. Les éléments de GWT....1547
65.4.1. Le compilateur....1547
65.4.2. JRE Emulation Library....1548
65.4.3. Les modules....1549
65.4.4. Les limitations....1549
65.5. L'interface graphique des applications GWT....1550
65.6. La personnalisation de l'interface....1551
65.7. Les composants (widgets)....1552
65.7.1. Les composants pour afficher des éléments....1554
65.7.1.1. Le composant Image....1554
65.7.1.2. Le composant Label....1555
65.7.1.3. Le composant DialogBox....1556
65.7.2. Les composants cliquables....1557
65.7.2.1. La classe Button....1557
65.7.2.2. La classe PushButton....1557
65.7.2.3. La classe ToggleButton....1558
65.7.2.4. La classe CheckBox....1558
65.7.2.5. La classe RadioButton....1559
65.7.2.6. Le composant HyperLink....1560
65.7.3. Les composants de saisie de texte....1560
65.7.3.1. Le composant TextBoxBase....1560
65.7.3.2. Le composant PasswordTextBox....1561
65.7.3.3. Le composant TextArea....1561
65.7.3.4. Le composant TextBox....1561
65.7.3.5. Le composant RichTextArea....1563
65.7.4. Les composants de sélection de données....1565
65.7.4.1. Le composant ListBox....1565
65.7.4.2. Le composant SuggestBox....1567
65.7.4.3. Le composant DateBox....1568
65.7.4.4. Le composant DatePicker....1569
65.7.5. Les composants HTML....1569
65.7.5.1. Le composant Frame....1569
65.7.5.2. Le composant HTML....1569
65.7.5.3. FileUpload....1570
65.7.5.4. Hidden....1570
65.7.6. Le composant Tree....1570
65.7.7. Les menus....1574
65.7.8. Le composant TabBar....1576
65.8. Les panneaux (panels)....1578
65.8.1. La classe Panel....1580
65.8.2. La classe RootPanel....1580
65.8.3. La classe SimplePanel....1581
65.8.4. La classe ComplexPanel....1581

Table des matières

65. GWT (Google Web Toolkit)
65.8.5. La classe FlowPanel....1582
65.8.6. La classe DeckPanel....1583
65.8.7. La classe TabPanel....1583
65.8.8. La classe FocusPanel....1584
65.8.9. La classe HTMLPanel....1584
65.8.10. La classe FormPanel....1584
65.8.11. La classe CellPanel....1585
65.8.12. La classe DockPanel....1585
65.8.13. La classe HorizontalPanel....1585
65.8.14. La classe VerticalPanel....1586
65.8.15. La classe CaptionPanel....1586
65.8.16. La classe PopupPanel....1587
65.8.17. La classe DialogBox....1588
65.8.18. La classe DisclosurePanel....1589
65.8.19. La classe AbsolutePanel....1589
65.8.20. La classe StackPanel....1589
65.8.21. La classe ScrollPanel....1590
65.8.22. La classe FlexTable....1590
65.8.23. La classe Frame....1591
65.8.24. La classe Grid....1592
65.8.25. La classe HorizontalSplitPanel....1593
65.8.26. La classe VerticalSplitPanel....1593
65.8.27. La classe HTMLTable....1594
65.8.28. La classe LazyPanel....1594
65.9. La création d'éléments réutilisables....1595
65.9.1. La création de composants personnalisés....1595
65.9.2. La création de modules réutilisables....1595
65.10. Les événements....1595
65.11. JSNI....1596
65.12. La configuration et l'internationalisation....1598
65.12.1. La configuration....1598
65.12.2. L'internationalisation....1599
65.13. L'appel de procédures distantes (Remote Procedure Call)....1601
65.13.1. GWT-RPC....1601
65.13.1.1. Une mise oeuvre avec un exemple simple....1601
65.13.1.2. La transmission d'objets lors des appels aux services....1608
65.13.1.3. L'invocation périodique d'un service....1612
65.13.2. L'objet RequestBuilder....1613
65.13.3. JavaScript Object Notation (JSON)....1613
65.14. La manipulation des documents XML....1613
65.15. La gestion de l'historique sur le navigateur....1615
65.16. Les tests unitaires....1615
65.17. Le déploiement d'une application....1616
65.18. Des composants tiers....1617
65.18.1. GWT-Dnd....1617
65.18.2. MyGWT....1617
65.18.3. GWT-Ext....1617
65.18.3.1. Installation et configuration....1617
65.18.3.2. La classe Panel....1618
65.18.3.3. La classe GridPanel....1619
65.19. Les ressources relatives à GWT....1622

Partie 10 : Les outils pour le développement....1623

66. Les outils du J.D.K....1625
66.1. Le compilateur javac....1625
66.1.1. La syntaxe de javac....1625
66.1.2. Les options de javac....1626

Table des matières

66. Les outils du J.D.K.

66.1.3. Les principales erreurs de compilation....1626
66.2. L'interpréteur java/javaw....1633
66.2.1. La syntaxe de l'outil java....1633
66.2.2. Les options de l'outil java....1634
66.3. L'outil jar....1634
66.3.1. L'intérêt du format jar....1634
66.3.2. La syntaxe de l'outil jar....1635
66.3.3. La création d'une archive jar....1636
66.3.4. Lister le contenu d'une archive jar....1636
66.3.5. L'extraction du contenu d'une archive jar....1637
66.3.6. L'utilisation des archives jar....1637
66.3.7. Le fichier manifest....1638
66.3.8. La signature d'une archive jar....1639
66.4. L'outil appletviewer pour tester des applets....1639
66.5. L'outil javadoc pour générer la documentation technique....1640
66.6. L'outil Java Check Update pour mettre à jour Java....1642
66.7. La base de données Java DB....1645
66.8. L'outil JConsole....1649

67. JavaDoc....1655

67.1. La mise en oeuvre....1655
67.2. Les tags définis par javadoc....1657
67.2.1. Le tag @author....1658
67.2.2. Le tag @deprecated....1658
67.2.3. Le tag @exception et @throws....1659
67.2.4. Le tag @param....1660
67.2.5. Le tag @return....1660
67.2.6. Le tag @see....1661
67.2.7. Le tag @since....1662
67.2.8. Le tag @version....1663
67.2.9. Le tag {@link}....1663
67.2.10. Le tag {@value}....1663
67.2.11. Le tag {@literal}....1664
67.2.12. Le tag {@linkplain}....1664
67.2.13. Le tag {@inheritDoc}....1664
67.2.14. Le tag {@docRoot}....1665
67.2.15. Le tag {@code}....1665
67.3. Un exemple....1665
67.4. Les fichiers pour enrichir la documentation des packages....1666
67.5. La documentation générée....1666

68. Les outils libres et commerciaux....1675

68.1. Les environnements de développements intégrés (IDE)....1675
68.1.1. Le projet Eclipse....1676
68.1.2. IBM Websphere Studio Application Developper....1677
68.1.3. IBM Rational Application Developer for WebSphere Software....1677
68.1.4. MyEclipse....1677
68.1.5. Netbeans....1677
68.1.6. Sun Java Studio Creator....1679
68.1.7. CodeGear (Borland) JBuilder....1680
68.1.8. JCreator....1681
68.1.9. Oracle JDeveloper....1681
68.1.10. IntelliJ IDEA....1682
68.1.11. BEA Workshop....1683
68.1.12. IBM Visual Age for Java....1683
68.1.13. Webgain Visual Café....1684
68.2. Les serveurs d'application....1684
68.2.1. JBoss Application Server....1684

Table des matières

68. Les outils libres et commerciaux
68.2.2. JOnAs....1684
68.2.3. GlassFish....1684
68.2.4. IBM Websphere Application Server....1685
68.2.5. BEA Weblogic....1685
68.2.6. Oracle Application Server....1685
68.2.7. Borland Entreprise Server....1685
68.2.8. Macromedia JRun....1685
68.3. Les conteneurs web....1686
68.3.1. Apache Tomcat....1686
68.3.2. Caucho Resin....1686
68.3.3. Enhydra....1686
68.4. Les conteneurs d'EJB....1686
68.4.1. OpenEJB....1686
68.5. Les outils divers....1687
68.5.1. Jikes....1687
68.5.2. GNU Compiler for Java....1687
68.5.3. Artistic Style....1690
68.6. Les MOM....1690
68.6.1. OpenJMS....1690
68.6.2. Joram....1692
68.6.3. OSMQ....1692
68.7. Les outils pour bases de données....1692
68.7.1. Derby....1692
68.7.2. SQuirrel-SQL....1692
68.8. Les outils de modélisation UML....1693
68.8.1. Argo UML....1693
68.8.2. Poseidon for UML....1693
68.8.3. StarUML....1693

69. Ant....1694
69.1. L'installation de l'outil Ant....1695
69.1.1. L'installation sous Windows....1695
69.2. Exécuter ant....1695
69.3. Le fichier build.xml....1696
69.3.1. Le projet....1696
69.3.2. Les commentaires....1697
69.3.3. Les propriétés....1697
69.3.4. Les ensembles de fichiers....1698
69.3.5. Les ensembles de motifs....1698
69.3.6. Les listes de fichiers....1699
69.3.7. Les éléments de chemins....1699
69.3.8. Les cibles....1699
69.4. Les tâches (task)....1700
69.4.1. echo....1701
69.4.2. mkdir....1702
69.4.3. delete....1703
69.4.4. copy....1704
69.4.5. tstamp....1704
69.4.6. java....1705
69.4.7. javac....1706
69.4.8. javadoc....1707
69.4.9. jar....1708

70. Maven....1709
70.1. L'installation....1709
70.2. Les plug-ins....1710
70.3. Le fichier project.xml....1710
70.4. L'exécution de Maven....1711

Table des matières

70. Maven
70.5. La génération du site du projet....1712
70.6. La compilation du projet....1714

71. Tomcat....1716
71.1. La présentation de Tomcat....1716
71.1.1. L'historique des versions....1716
71.2. L'installation....1717
71.2.1. L'installation de Tomcat 3.1 sous Windows 98....1717
71.2.2. L'installation de Tomcat 4.0 sur Windows 98....1718
71.2.3. L'installation de Tomcat 5.0 sur Windows....1720
71.2.4. L'installation de Tomcat 5.5 sous Windows avec l'installer....1720
71.2.5. L'installation Tomcat 6.0 sous Windows avec l'installer....1724
71.2.6. La structure des répertoires....1724
71.2.6.1. 1.2.6.1 La structure des répertoires de Tomcat 4....1724
71.2.6.2. La structure des répertoires de Tomcat 5....1725
71.2.6.3. La structure des répertoires de Tomcat 6....1725
71.3. L'exécution de Tomcat....1726
71.3.1. L'exécution sous Windows de Tomcat 4.0....1726
71.3.2. L'exécution sous Windows de Tomcat 5.0....1726
71.3.3. La vérification de l'exécution....1729
71.4. L'architecture....1730
71.4.1. Les connecteurs....1731
71.4.2. Les services....1732
71.5. La configuration....1732
71.5.1. Le fichier server.xml....1732
71.5.1.1. Le fichier server.xml avec Tomcat 5....1733
71.5.1.2. Les valves....1735
71.5.2. La gestion des rôles....1735
71.6. L'outil Tomcat Administration Tool....1736
71.6.0.1. La gestion des utilisateurs, des rôles et des groupes....1738
71.6.1. La création d'une DataSource dans Tomcat....1739
71.7. Le déploiement des applications WEB....1740
71.7.1. Déployer une application web avec Tomcat 5....1740
71.7.1.1. Déployer une application au lancement de Tomcat....1740
71.7.1.2. Déployer une application sur Tomcat en cours d'exécution....1741
71.7.1.3. L'utilisation d'un contexte....1741
71.7.1.4. Déployer une application avec le Tomcat Manager....1741
71.7.1.5. Déployer une application avec les tâches Ant du Manager....1741
71.7.1.6. Déployer une application avec le TCD....1741
71.8. Tomcat pour le développeur....1741
71.8.1. Accéder à une ressource par son url....1741
71.8.2. La structure d'une application web et format war....1742
71.8.3. La configuration d'un contexte....1743
71.8.3.1. La configuration d'un contexte avec Tomcat 5....1743
71.8.4. L'invocation dynamique de servlets....1743
71.8.5. Les bibliothèques partagées....1745
71.8.5.1. Les bibliothèques partagées sous Tomcat 5....1745
71.9. Le gestionnaire d'applications (Tomcat manager)....1746
71.9.1. L'utilisation de l'interface graphique....1746
71.9.1.1. Le déploiement d'une application....1749
71.9.1.2. La gestion des applications....1750
71.9.2. L'utilisation des commandes par requêtes HTTP....1751
71.9.2.1. La commande list....1751
71.9.2.2. La commande serverinfo....1752
71.9.2.3. La commande reload....1752
71.9.2.4. La commande resources....1752
71.9.2.5. La commande roles....1753
71.9.2.6. La commande sessions....1753

Table des matières

71. Tomcat
71.9.2.7. La commande stop....1753
71.9.2.8. La commande start....1754
71.9.2.9. La commande undeploy....1754
71.9.2.10. La commande deploy....1755
71.9.3. L'utilisation du manager avec des tâches Ant....1755
71.9.4. L'utilisation de la servlet JMXProxy....1757
71.10. L'outil Tomcat Client Deployer....1758
71.11. Les optimisations....1759
71.12. La sécurisation du serveur....1760

72. Des outils open source pour faciliter le développement....1761
72.1. CheckStyle....1761
72.1.1. L'installation....1761
72.1.2. L'utilisation avec Ant....1762
72.1.3. L'utilisation en ligne de commandes....1765
72.2. Jalopy....1765
72.2.1. L'utilisation avec Ant....1766
72.2.2. Les conventions....1768
72.3. XDoclet....1770
72.4. Middlegen....1770

Partie 11 : Concevoir et développer des applications....1771

73. Java et UML....1773
73.1. La présentation d'UML....1773
73.2. Les commentaires....1774
73.3. Les cas d'utilisation (uses cases)....1774
73.4. Le diagramme de séquence....1776
73.5. Le diagramme de collaboration....1776
73.6. Le diagramme d'états-transitions....1777
73.7. Le diagramme d'activités....1778
73.8. Le diagramme de classes....1778
73.9. Le diagramme d'objets....1780
73.10. Le diagramme de composants....1781
73.11. Le diagramme de déploiement....1781

74. Les motifs de conception (design patterns)....1782
74.1. Les modèles de création....1782
74.1.1. Fabrique (Factory)....1782
74.1.2. Fabrique abstraite (abstract Factory)....1786
74.1.3. Monteur (Builder)....1789
74.1.4. Prototype (Prototype)....1790
74.1.5. Singleton (Singleton)....1790
74.2. Les modèles de structuration....1791
74.2.1. Façade (Facade)....1792
74.2.2. Décorateur (Decorator)....1796
74.3. Les modèles de comportement....1800

75. Des normes de développement....1801
75.1. Les fichiers....1801
75.1.1. Les packages....1801
75.1.2. Le nom de fichiers....1802
75.1.3. Le contenu des fichiers sources....1802
75.1.4. Les commentaires de début de fichier....1802
75.1.5. Les clauses concernant les packages....1803
75.1.6. La déclaration des classes et des interfaces....1803
75.2. La documentation du code....1803
75.2.1. Les commentaires de documentation....1803

Table des matières

75. Des normes de développement
75.2.1.1. L'utilisation des commentaires de documentation....1803
75.2.1.2. Les commentaires pour une classe ou une interface....1804
75.2.1.3. Les commentaires pour une variable de classe ou d'instance....1804
75.2.1.4. Les commentaires pour une méthode....1805
75.2.2. Les commentaires de traitements....1805
75.2.2.1. Les commentaires sur une ligne....1805
75.2.2.2. Les commentaires sur une portion de ligne....1806
75.2.2.3. Les commentaires multi-lignes....1806
75.2.2.4. Les commentaires de fin de ligne....1806
75.3. Les déclarations....1807
75.3.1. La déclaration des variables....1807
75.3.2. La déclaration des classes et des méthodes....1808
75.3.3. La déclaration des constructeurs....1808
75.3.4. Les conventions de nommage des entités....1809
75.4. Les séparateurs....1810
75.4.1. L'indentation....1810
75.4.2. Les lignes blanches....1811
75.4.3. Les espaces....1811
75.4.4. La coupure de lignes....1812
75.5. Les traitements....1812
75.5.1. Les instructions composées....1812
75.5.2. L'instruction return....1812
75.5.3. L'instruction if....1812
75.5.4. L'instruction for....1813
75.5.5. L'instruction while....1813
75.5.6. L'instruction do-while....1813
75.5.7. L'instruction switch....1814
75.5.8. Les instructions try-catch....1814
75.6. Les règles de programmation....1814
75.6.1. Le respect des règles d'encapsulation....1814
75.6.2. Les références aux variables et méthodes de classes....1815
75.6.3. Les constantes....1815
75.6.4. L'assignement des variables....1815
75.6.5. L'usage des parenthèses....1815
75.6.6. La valeur de retour....1816
75.6.7. La codification de la condition dans l'opérateur ternaire ? :....1816
75.6.8. La déclaration d'un tableau....1816

76. L'encodage des caractères....1817
76.1. L'utilisation des caractères dans la JVM....1817
76.1.1. Le stockage des caractères dans la JVM....1817
76.1.2. L'encodage des caractères par défaut....1817
76.2. Les jeux d'encodage de caractères....1818
76.3. Unicode....1818
76.3.1. L'encodage des caractères Unicode....1818
76.3.2. Le marqueur optionnel BOM....1819
76.4. L'encodage de caractères....1820
76.4.1. Les classes du package java.lang....1821
76.4.2. Les classes du package java.io....1821
76.4.3. Le package java.nio....1822
76.5. L'encodage du code source....1822
76.6. L'encodage de caractères avec différentes technologies....1823
76.6.1. L'encodage de caractères dans les fichiers....1823
76.6.2. L'encodage de caractères dans une application web....1824
76.6.3. L'encodage de caractères avec JDBC....1824

Table des matières

77. Les frameworks....1825
77.1. La présentation des concepts....1825
77.1.1. La définition d'un framework....1825
77.1.2. L'utilité de mettre en oeuvre des frameworks....1826
77.1.3. Les différentes catégories de framework....1827
77.1.4. Les socles techniques....1827
77.2. Les frameworks pour les applications web....1828
77.3. L'architecture pour les applications web....1828
77.3.1. Le modèle MVC....1828
77.4. Le modèle MVC type 1....1829
77.5. Le modèle MVC de type 2....1829
77.5.1. Les différents types de framework web....1830
77.5.2. Des frameworks pour le développement web....1831
77.5.2.1. Apache Struts....1831
77.5.2.2. Spring MVC....1832
77.5.2.3. Webwork....1832
77.5.2.4. Tapestry....1832
77.5.2.5. Java Server Faces....1832
77.5.2.6. Struts 2....1833
77.5.2.7. Struts Shale....1833
77.5.2.8. Expresso....1833
77.5.2.9. Jena....1834
77.5.2.10. Echo 2....1834
77.5.2.11. Barracuda....1834
77.5.2.12. Stripes....1834
77.5.2.13. Turbine....1834
77.6. Les frameworks de mapping Objet/Relationel....1834
77.7. Les frameworks de logging....1834

78. Les frameworks de tests....1836
78.1. Les tests unitaires....1836
78.1.1. L'utilité des tests unitaires automatisés....1837
78.1.2. Quelques principes pour mettre en oeuvre de tests unitaires....1838
78.1.3. Les difficultés lors de la mise en oeuvre de tests unitaires....1839
78.1.4. Des best practices....1840
78.2. Les frameworks et outils de tests....1841
78.2.1. Les frameworks pour les tests unitaires....1841
78.2.2. Les frameworks pour le mocking....1841
78.2.3. Les extensions de JUnit....1841
78.2.4. Les outils de tests de charge....1842
78.2.5. Les outils d'analyse de couverture de tests....1842

79. JUnit....1843
79.1. Un exemple très simple....1844
79.2. L'écriture des cas de tests....1845
79.2.1. La définition de la classe de tests....1846
79.2.2. La définition des cas de tests....1847
79.2.3. L'initialisation des cas de tests....1849
79.2.4. Le test de la levée d'exceptions....1852
79.2.5. L'héritage d'une classe de base....1853
79.3. L'exécution des tests....1854
79.3.1. L'exécution des tests dans la console....1854
79.3.2. L'exécution des tests dans une application graphique....1855
79.3.3. L'exécution d'une classe de tests....1857
79.3.4. L'exécution répétée d'un cas de test....1857
79.3.5. L'exécution concurrente de tests....1858
79.4. Les suites de tests....1859
79.5. L'automatisation des tests avec Ant....1860
79.6. JUnit 4....1861

Table des matières

79. JUnit

79.6.1. La définition d'une classe de tests....1862
79.6.2. La définition des cas de tests....1862
79.6.3. L'initialisation des cas des tests....1863
79.6.4. Le test de la levée d'exceptions....1863
79.6.5. L'exécution des tests....1864
79.6.6. Un exemple de migration de JUnit 3 vers JUnit 4....1864
79.6.7. La limitation du temps d'exécution d'un cas de test....1866
79.6.8. Les tests paramétrés....1867
79.6.9. La rétro compatibilité....1867
79.6.10. L'organisation des tests....1868

80. Les objets de type Mock....1871
80.1. Les doublures d'objets et les objets de type mock....1871
80.1.1. Les types d'objets mock....1872
80.1.2. Exemple d'utilisation dans les tests unitaires....1872
80.1.3. La mise en oeuvre des objets de type mock....1873
80.2. L'utilité des objets de type mock....1873
80.2.1. L'utilisation dans les tests unitaires....1873
80.2.2. L'utilisation dans les tests d'intégration....1874
80.2.3. La simulation de l'appel à des ressources....1874
80.2.4. La simulation du comportement de composants ayant des résultats variables....1874
80.2.5. La simulation des cas d'erreurs....1874
80.3. Les tests unitaires et les dépendances....1875
80.4. L'obligation d'avoir une bonne organisation du code....1875
80.4.1. Quelques recommandations....1875
80.4.2. Les dépendances et les tests unitaires....1876
80.4.3. Exemple de mise en oeuvre de l'injection de dépendances....1877
80.4.4. Limiter l'usage des singletons....1879
80.4.5. Encapsuler les ressources externes....1879
80.5. Les frameworks....1879
80.5.1. EasyMock....1881
80.5.1.1. La création d'objets mock....1882
80.5.1.2. La définition du comportement des objets mock....1883
80.5.1.3. L'initialisation des objets mock....1884
80.5.1.4. La vérification des invocations des objets mock....1884
80.5.1.5. La vérification de l'ordre d'invocations des mocks....1885
80.5.1.6. La gestion de plusieurs objets de type mock....1886
80.5.1.7. Un exemple complexe....1886
80.5.2. Mockito....1887
80.5.3. JMock....1887
80.5.4. MockRunner....1887
80.6. Les outils pour générer des objets mock....1888
80.6.1. MockObjects....1888
80.6.2. MockMaker....1888
80.7. Les inconvénients des objets de type mock....1888

81. Des bibliothèques open source....1889
81.1. JFreeChart....1889
81.2. Beanshell....1893
81.3. Jakarta Commons....1893
81.4. Jakarta ORO....1894
81.5. Joda Time....1894
81.6. Quartz....1894
81.7. JGoodies....1894
81.8. Apache Lucene....1894

Table des matières

82. La génération de documents....1895
82.1. Apache POI....1895
82.1.1. POI-HSSF....1896
82.1.1.1. L'API de type usermodel....1896
82.1.1.2. L'API de type eventusermodel....1914
82.2. iText....1914
82.2.1. Un exemple très simple....1915
82.2.2. L'API de iText....1915
82.2.3. La création d'un document....1916
82.2.3.1. La classe Document....1916
82.2.3.2. Les objets de type DocWriter....1919
82.2.4. L'ajout de contenu au document....1921
82.2.4.1. Les polices de caractères....1921
82.2.4.2. Le classe Chunk....1925
82.2.4.3. La classe Phrase....1927
82.2.4.4. La classe Paragraph....1929
82.2.4.5. La classe Chapter....1931
82.2.4.6. La classe Section....1932
82.2.4.7. La création d'une nouvelle page....1933
82.2.4.8. La classe Anchor....1934
82.2.4.9. Les classes List et ListItem....1935
82.2.4.10. La classe Table....1936
82.2.5. Des fonctionnalités avancées....1940
82.2.5.1. Insérer une image....1940

83. La validation des données....1941
83.1. Quelques recommandations sur la validation des données....1942
83.2. L'API Bean Validation (JSR 303)....1942
83.2.1. La présentation de l'API....1943
83.2.1.1. Les objectifs de l'API....1943
83.2.1.2. Les éléments et concepts utilisés par l'API....1943
83.2.1.3. Les contraintes et leur validation avec l'API....1944
83.2.1.4. La mise en oeuvre générale de l'API....1944
83.2.1.5. Un exemple simple de mise en oeuvre....1945
83.2.2. La déclaration des contraintes....1947
83.2.2.1. La déclaration des contraintes sur les champs....1947
83.2.2.2. La déclaration des contraintes sur les propriétés....1948
83.2.2.3. La déclaration des contraintes sur une classe....1949
83.2.2.4. L'héritage de contraintes....1949
83.2.2.5. Les contraintes de validation d'un ensemble d'objets....1950
83.2.3. La validation des contraintes....1953
83.2.3.1. L'obtention d'un valideur....1953
83.2.3.2. L'interface Validator....1954
83.2.3.3. L'utilisation d'un valideur....1954
83.2.3.4. L'interface ConstraintViolation....1956
83.2.3.5. La mise en oeuvre des groupes....1957
83.2.3.6. Définir et utiliser un groupe implicite....1959
83.2.3.7. La définition de l'ordre des validations....1961
83.2.3.8. La redéfinition du groupe par défaut....1962
83.2.4. Les contraintes standards....1962
83.2.4.1. L'annotation @Null....1963
83.2.4.2. L'annotation @NotNull....1964
83.2.4.3. L'annotation @AssertTrue....1964
83.2.4.4. L'annotation @AssertFalse....1965
83.2.4.5. L'annotation @Min....1966
83.2.4.6. L'annotation @Max....1966
83.2.4.7. L'annotation @DecimalMin....1967
83.2.4.8. L'annotation @DecimalMax....1968
83.2.4.9. L'annotation @Size....1969

Table des matières

83. La validation des données
83.2.4.10. L'annotation @Digits....1970
83.2.4.11. L'annotation @Past....1970
83.2.4.12. L'annotation @Future....1971
83.2.4.13. L'annotation @Pattern....1972
83.2.5. Le développement de contraintes personnalisées....1972
83.2.5.1. La création de l'annotation....1972
83.2.5.2. La création de la classe de validation....1976
83.2.5.3. Le message d'erreur....1979
83.2.5.4. L'utilisation d'une contrainte....1980
83.2.5.5. Application multiple d'une contrainte....1981
83.2.6. Les contraintes composées....1982
83.2.7. L'interpolation des messages....1986
83.2.7.1. L'algorithme d'une interpolation par défaut....1986
83.2.7.2. Le développement d'un MessageInterpolator spécifique....1986
83.2.8. Bootstrapping....1987
83.2.8.1. L'utilisation du Java Service Provider....1988
83.2.8.2. L'utilisation de la classe Validation....1988
83.2.8.3. Les interfaces ValidationProvider et ValidationProviderResolver....1990
83.2.8.4. L'interface MessageInterpolator....1990
83.2.8.5. L'interface TraversableResolver....1991
83.2.8.6. L'interface ConstraintValidatorFactory....1991
83.2.8.7. L'interface ValidatorFactory....1992
83.2.8.8. L'interface Configuration....1992
83.2.8.9. Le fichier de configuration META-INF/validation.xml....1993
83.2.9. La définition de contraintes dans un fichier XML....1994
83.2.10. L'API de recherche des contraintes....1994
83.2.10.1. L'interface ElementDescriptor....1994
83.2.10.2. L'interface ConstraintFinder....1995
83.2.10.3. L'interface BeanDescriptor....1995
83.2.10.4. L'interface PropertyDescriptor....1995
83.2.10.5. L'interface ConstraintDescriptor....1996
83.2.10.6. Un exemple de mise en oeuvre....1996
83.2.11. La validation des paramètres et de la valeur de retour d'une méthode....1997
83.2.12. L'implémentation de référence : Hibernate Validator....1998
83.2.13. Les avantages et les inconvénients....1998
83.3. D'autres frameworks pour la validation des données....1998

84. La communauté Java....2000
84.1. Le JCP....2000
84.2. Les ressources proposées par Sun....2001
84.3. Le programme Sun Developer Network....2001
84.3.1. SDN share....2001
84.4. La communauté Java.net....2001
84.5. Les JUG....2002
84.6. D'autres User Groups....2003
84.7. Java BlackBelt....2004
84.8. Les conférences....2004
84.8.1. JavaOne....2004
84.8.2. Devoxx (ex : JavaPolis)....2005
84.8.3. Jazoon....2005
84.9. Webographie....2005
84.10. Les communautés open source....2007
84.10.1. Apache - Jakarta....2007
84.10.2. Codehaus....2007
84.10.3. OW2....2007
84.10.4. JBoss....2007
84.10.5. Source Forge....2007

Table des matières

Partie 12 : Développement d'applications mobiles 2009

85. J2ME / Java ME 2010
- 85.1. L'historique de la plate-forme 2010
- 85.2. La présentation de J2ME / Java ME 2011
- 85.3. Les configurations 2012
- 85.4. Les profiles 2013
- 85.5. J2ME Wireless Toolkit 1.0.4 2014
  - 85.5.1. L'installation du J2ME Wireless Toolkit 1.0.4 2014
  - 85.5.2. Les premiers pas 2015
- 85.6. J2ME wireless toolkit 2.1 2018
  - 85.6.1. L'installation du J2ME Wireless Toolkit 2.1 2018
  - 85.6.2. Les premiers pas 2018

86. CLDC 2024
- 86.1. Le package java.lang 2024
- 86.2. Le package java.io 2025
- 86.3. Le package java.util 2026
- 86.4. Le package javax.microedition.io 2026

87. MIDP 2027
- 87.1. Les Midlets 2027
- 87.2. L'interface utilisateur 2028
  - 87.2.1. La classe Display 2029
  - 87.2.2. La classe TextBox 2030
  - 87.2.3. La classe List 2031
  - 87.2.4. La classe Form 2032
  - 87.2.5. La classe Item 2033
  - 87.2.6. La classe Alert 2034
- 87.3. La gestion des événements 2035
- 87.4. Le stockage et la gestion des données 2036
  - 87.4.1. La classe RecordStore 2036
- 87.5. Les suites de midlets 2037
- 87.6. Packager une midlet 2038
  - 87.6.1. Le fichier manifest 2038
- 87.7. MIDP for Palm O.S. 2038
  - 87.7.1. L'installation 2038
  - 87.7.2. La création d'un fichier .prc 2039
  - 87.7.3. L'installation et l'exécution d'une application 2042

88. CDC 2043

89. Les profils du CDC 2044
- 89.1. Foundation profile 2044
- 89.2. Le Personal Basis Profile (PBP) 2044
- 89.3. Le Personal Profile (PP) 2044

90. Les autres technologies pour les applications mobiles 2046
- 90.1. KJava 2046
- 90.2. PDAP (PDA Profile) 2046
- 90.3. PersonalJava 2047
- 90.4. Java Phone 2047
- 90.5. JavaCard 2047
- 90.6. Embedded Java 2047
- 90.7. Waba, Super Waba, Visual Waba 2047

Partie 13 : Annexes 2048
- Annexe A : GNU Free Documentation License 2048
- Annexe B : Glossaire 2052

Version 1.40

du 10/08/2010

par Jean Michel DOUDOUX

Préambule

A propos de ce document

L'idée de départ de ce document était de prendre des notes relatives à mes premiers essais avec Java en 1996. Ces notes ont tellement grossies que j'ai décidé de les formaliser un peu plus et de les diffuser sur internet d'abord sous la forme d'articles puis rassemblées pour former le présent didacticiel.

Aujourd'hui, celui-ci est composé de treize grandes parties :

1. les bases du langage java
2. le développement des interfaces graphiques
3. les API avancées
4. l'utilisation de documents XML
5. l'accès aux bases de données
6. la machine virtuelle Java (JVM)
7. le développement d'applications d'entreprises
8. le développement d'applications web
9. le développement d'applications RIA/RDA
10. les outils de développement
11. la conception et le développement des applications
12. le développement d'applications mobiles
13. les annexes

Chacune de ces parties est composée de plusieurs chapitres dont voici la liste complète :

- Préambule
- Présentation de Java
- Les notions et techniques de base en Java
- La syntaxe et les éléments de bases de Java
- POO avec Java
- Les packages de base
- Les fonctions mathématiques
- La gestion des exceptions
- Le multitâche
- J2SE 5.0 (JDK 1.5 : nom de code Tiger)
- Annotations
- Le graphisme en java
- Les éléments d'interfaces graphiques de l'AWT
- La création d'interfaces graphiques avec AWT
- L'interception des actions de l'utilisateur
- Le développement d'interfaces graphiques avec SWING
- Le développement d'interfaces graphiques avec SWT
- JFace
- Les collections
- Les flux
- La sérialisation
- L'interaction avec le réseau
- La gestion dynamique des objets et l'introspection
- L'appel de méthode distantes : RMI
- L'internationalisation
- Les composants Java beans
- Logging
- La sécurité
- JNI (Java Native Interface)
- JNDI (Java Naming and Directory Interface)
- Scripting
- JMX (Java Management Extensions)
- Java et XML

- SAX (Simple API for XML)
- DOM (Document Object Model)
- XSLT (Extensible Stylesheet Language Transformations)
- Les modèles de document
- JAXB (Java Architecture for XML Binding)
- StAX (Streaming Api for XML)
- La persistance des objets
- JDBC (Java DataBase Connectivity)
- JDO (Java Data Object)
- Hibernate
- JPA (Java Persistence API)
- La JVM (Java Virtual Machine)
- La gestion de la mémoire
- La décompilation et l'obfuscation
- Terracotta
- Java 2 Entreprise Edition
- JavaMail
- JMS (Java Messaging Service)
- Les EJB (Entreprise Java Bean)
- Les EJB 3
- Les EJB 3.1
- Les services web de type Soap
- Les servlets
- Les JSP (Java Server Pages)
- JSTL (Java server page Standard Tag Library)
- Struts
- JSF (Java Server Faces)
- D'autres frameworks pour les applications web
- Les applications riches : RIA et RDA
- Les applets en java
- Java Web Start (JWS)
- Ajax
- GWT (Google Web Toolkit)
- Les outils du J.D.K.
- JavaDoc
- Les outils libres et commerciaux
- Ant
- Maven
- Tomcat
- Des outils open source
- Java et UML
- Les motifs de conception (design patterns)
- Des normes de développement
- L'encodage des caractères
- Les frameworks
- Les frameworks de tests
- JUnit
- Les objets de type mock
- Des bibliothèques open source
- La génération de documents
- La validation des données
- La communauté Java
- J2ME
- CLDC
- MIDP
- CDC
- Les profils du CDC
- Les autres technologies

Je souhaiterais l'enrichir pour qu'il couvre un maximum de sujets autour du développement avec les technologies relatives à Java. Ce souhait est ambitieux car les API de Java et open source sont très riches et ne cessent de s'enrichir au fil des versions et des années.

Dans chaque chapitre, les classes et les membres des classes décrites ne le sont que partiellement : le but n'est pas de remplacer la documentation d'une API mais de faciliter ces premières mises en oeuvre. Ainsi pour une description complète de chaque classe, il faut consulter la documentation fournie par Sun au format HTML pour les API du JDK et la documentation fournie par les fournisseurs respectifs des autres API tiers.

Je suis ouvert à toutes réactions ou suggestions concernant ce document notamment le signalement des erreurs, les points à éclaircir, les sujets à ajouter, etc. ... N'hésitez pas à me contacter : jean-michel.doudoux@wanadoo.fr

La dernière version publiée de ce document est disponible aux formats HTML et PDF sur mon site perso : http://www.jmdoudoux.fr/java/

Il est aussi disponible dans les deux formats à l'url : http://jmdoudoux.developpez.com/cours/developpons/java/

Ce manuel est fourni en l'état, sans aucune garantie. L'auteur ne peut être tenu pour responsable des éventuels dommages causés par l'utilisation des informations fournies dans ce document.

La version pdf de ce document est réalisée grâce à l'outil HTMLDOC version 1.8.23 de la société Easy Software Products. Cet excellent outil freeware peut être téléchargé à l'adresse : http://www.easysw.com

La version sur mon site perso utilise deux très bons outils open source :

- SyntaxHighlighter : pour afficher et appliquer une coloration syntaxique des exemples (http://alexgorbatchev.com/)
- JavasScript Tree Menu : pour afficher l'arborescence des chapitres et sections et faciliter la navigation dans ce document (http://www.treeview.net)

Remerciements

Je tiens à remercier les personnes qui m'ont apporté leur soutien au travers de courriers électroniques de remerciements, de félicitations ou d'encouragements.

Je tiens aussi particulièrement à exprimer ma gratitude aux personnes qui m'ont fait part de correctifs ou d'idées d'évolutions : ainsi pour leurs actions, je tiens particulièrement à remercier Vincent Brabant, Thierry Durand et David Riou.

Notes de licence

Copyright (C) 1999-2010 DOUDOUX Jean Michel

Vous pouvez copier, redistribuer et/ou modifier ce document selon les termes de la Licence de Documentation Libre GNU, Version 1.1 ou toute autre version ultérieure publiée par la Free Software Foundation; les Sections Invariantes étant constitués du chapitre Préambule, aucun Texte de Première de Couverture, et aucun Texte de Quatrième de Couverture. Une copie de la licence est incluse dans la section GNU FreeDocumentation Licence de ce document.

La version la plus récente de cette licence est disponible à l'adresse : http://www.fsf.org/copyleft/fdl.html.

Marques déposées

Sun, Sun Microsystems, le logo Sun et Java sont des marques déposées de Sun Microsystems Inc jusqu'en décembre 2009 puis d'Oracle à partir de janvier 2010.

Les autres marques et les noms de produits cités dans ce document sont la propriété de leur éditeur respectif.

Historique des versions

Version	Date	Evolutions
0.10	15/01/2001	brouillon : 1ere version diffusée sur le web.
0.20	11/03/2001	ajout des chapitres JSP et serialization, des informations sur le JDK et son installation, corrections diverses.
0.30	10/05/2001	ajout des chapitres flux, beans et outils du JDK, corrections diverses.
0.40	10/11/2001	réorganisation des chapitres et remise en page du document au format HTML (1 page par chapitre) pour faciliter la maintenance ajout des chapitres : collections, XML, JMS, début des chapitres Swing et EJB séparation du chapitre AWT en trois chapitres.
0.50	31/04/2002	séparation du document en trois parties ajout des chapitres : logging, JNDI, Java mail, services web, outils du JDK, outils lires et commerciaux, Java et UML, motifs de conception compléments ajoutés aux chapitres : JDBC, Javadoc, intéraction avec le réseau, Java et xml, bibliothèques de classes
0.60	23/12/2002	ajout des chapitres : JSTL, JDO, Ant, les frameworks ajout des sections : Java et MySQL, les classes internes, les expressions régulières, dom4j compléments ajoutés aux chapitres : JNDI, design patterns, J2EE, EJB
0.65	05/04/2003	ajout d'un index sous la forme d'un arbre hiérarchique affiché dans un frame de la version HTML ajout des sections : DOM, JAXB, bibliothèques de tags personnalisés, package .war compléments ajoutés aux chapitres : EJB, réseau, services web
0.70	05/07/2003	ajout de la partie sur le développement d'applications mobiles contenant les chapitres : J2ME, CLDC, MIDP, CDC, Personal Profile, les autres technologies ajout des chapitres : le multitâche, les frameworks de tests, la sécurité, les frameworks pour les app web compléments ajoutés aux chapitres : JDBC, JSP, servlets, intéraction avec le réseau application d'une feuille de styles CSS pour la version HTML corrections et ajouts divers (652 pages)
0.75	21/03/2004	ajout des chapitres : le développement d'interfaces avec SWT, Java Web Start, JNI compléments ajoutés aux chapitres : GCJ, JDO, nombreuses corrections et ajouts divers notamment dans les premiers chapitres (737 pages)
0.80 0.80.1 0.80.2	29/06/2004 15/10/2004 02/11/2004	ajout des chapitres : le JDK 1.5, des bibliothèques open source, des outils open source, Maven et d'autres solutions de mapping objet-relationnel ajout des sections : Installation J2SE 1.4.2 sous Windows, J2EE 1.4 SDK, J2ME WTK 2.1 compléments ajoutés aux chapitres : Ant, Jdbc, Swing, Java et UML, MIDP, J2ME, JSP, JDO nombreuses corrections et ajouts divers (831 pages)

0.85	27/11/2005	ajout du chapitre : Java Server Faces ajout des sections : java updates, le composant JTree nombreuses corrections et ajouts divers (922 pages)
0.90	11/09/2006	ajout des chapitres : Ajax, Frameworks, Struts compléments ajoutés aux chapitres : Javadoc, JNDI, Design pattern (façade, fabrique, fabrique abstraite), JFace nombreuses corrections et ajouts divers (1092 pages)
0.95 0.95.1 0.95.2	18/11/2007 12/06/2008 01/11/2008	ajout des parties : utilisation de documents XML, l'Accès aux bases de données, développement d'applications web, concevoir et développer des applications ajout des chapitres : Scripting, persistance des objets, StAX, JPA, Tomcat compléments ajoutés aux chapitres : Java SE 6, JAXB 2.0, Java DB, Java EE 5, JMS 1.1, OpenJMS, les menus avec Swing, le design pattern décorateur, Rowset nombreuses corrections et ajouts divers (1305 pages)
1.00	16/03/2009	ajout des parties : la JVM et le développement d'applications RIA/RDA ajout des chapitres : annotations, décompilation et obfuscation, génération de documents, GWT, JVM, la gestion de la mémoire, la communauté Java, les applications RIA/RDA réécriture complète des chapitres : les modèles de documents (JDOM), les techniques de bases, Logging (log4j), Junit compléments ajoutés aux chapitres : la gestion des exceptions, JDBC (performances), les fonctions mathématiques (BigDecimal), JDBC 3.0, les framework de tests, Apache POI, iText nombreuses corrections et ajouts divers (1672 pages)
1.10	04/08/2009	ajout des chapitres : JMX, EJB 3 et l'encodage des caractères compléments ajoutés aux chapitres : J2ME /Java ME corrections et ajouts divers (1820 pages)
1.20	29/10/2009	ajout du chapitre : les objets de type mock ajout de la section : Java 6 Update compléments ajoutés aux chapitres : les frameworks de tests, JUnit, GWT et RIA nombreuses corrections et ajouts divers (1888 pages)
1.30	27/03/2010	ajout des chapitres : la validation de données, EJB 3.1 ajout de la section : Java EE 6 réécriture du chapitre : les services web de type Soap nombreuses corrections et ajouts divers (2035 pages)
1.40	08/08/2010	ajout du chapitre : Terracotta ajout de la section : Hibernate (les relations 1/1) réécriture de la section : les énumérations

		nombreuses corrections et ajouts divers (2105 pages)

Partie 1 : les bases du langage Java

Cette première partie est chargée de présenter les bases du langage java.

Elle comporte les chapitres suivants :

- Présentation de Java : introduit le langage Java en présentant les différentes éditions et versions du JDK, les caractéristiques du langage et décrit l'installation du JDK
- Les notions et techniques de base en Java : présente rapidement quelques notions de base et comment compiler et exécuter une application
- La syntaxe et les éléments de bases de Java : explore les éléments du langage d'un point de vue syntaxique
- La programmation orientée objet : explore comment Java utilise et permet d'utiliser la programmation orientée objet
- Les packages de bases : propose une présentation rapide des principales API fournies avec le JDK
- Les fonctions mathématiques : indique comment utiliser les fonctions mathématiques
- La gestion des exceptions : explore la faculté de Java pour traiter et gérer les anomalies qui surviennent lors de l'exécution du code
- Le multitâche : présente et met en oeuvre les mécanismes des threads qui permettent de répartir différents traitements d'un même programme en plusieurs unités distinctes pour permettre leur exécution "simultanée"
- JDK 1.5 (nom de code Tiger) : détaille les nouvelles fonctionnalités du langage de la version 1.5
- Les annotations : présente les annotations qui sont des méta données insérées dans le code source et leurs mises en oeuvre.

1. Présentation de Java

Java est un langage de programmation à usage général, évolué et orienté objet dont la syntaxe est proche du C.

Ce chapitre contient plusieurs sections :

- Les caractéristiques
- Un bref historique de Java
- Les différentes éditions et versions de Java
- Un rapide tour d'horizon des API et de quelques outils
- Les différences entre Java et JavaScript
- L'installation du JDK

1.1. Les caractéristiques

Java possède un certain nombre de caractéristiques qui ont largement contribué à son énorme succès :

Java est interprété	le source est compilé en pseudo code ou byte code puis exécuté par un interpréteur Java : la Java Virtual Machine (JVM). Ce concept est à la base du slogan de Sun pour Java : WORA (Write Once, Run Anywhere : écrire une fois, exécuter partout). En effet, le byte code, s'il ne contient pas de code spécifique à une plate-forme particulière peut être exécuté et obtenir quasiment les même résultats sur toutes les machines disposant d'une JVM.
Java est portable : il est indépendant de toute plate-forme	il n'y a pas de compilation spécifique pour chaque plate forme. Le code reste indépendant de la machine sur laquelle il s'exécute. Il est possible d'exécuter des programmes Java sur tous les environnements qui possèdent une Java Virtual Machine. Cette indépendance est assurée au niveau du code source grâce à Unicode et au niveau du byte code.
Java est orienté objet.	comme la plupart des langages récents, Java est orienté objet. Chaque fichier source contient la définition d'une ou plusieurs classes qui sont utilisées les unes avec les autres pour former une application. Java n'est pas complètement objet car il définit des types primitifs (entier, caractère, flottant, booléen,...).
Java est simple	le choix de ses auteurs a été d'abandonner des éléments mal compris ou mal exploités des autres langages tels que la notion de pointeurs (pour éviter les incidents en manipulant directement la mémoire), l'héritage multiple et la surcharge des opérateurs, ...
Java est fortement typé	toutes les variables sont typées et il n'existe pas de conversion automatique qui risquerait une perte de données. Si une telle conversion doit être réalisée, le développeur doit obligatoirement utiliser un cast ou une méthode statique fournie en standard pour la réaliser.
Java assure la gestion de la mémoire	l'allocation de la mémoire pour un objet est automatique à sa création et Java récupère automatiquement la mémoire inutilisée grâce au garbage collector qui restitue les zones de mémoire laissées libres suite à la destruction des objets.

Java est sûre	la sécurité fait partie intégrante du système d'exécution et du compilateur. Un programme Java planté ne menace pas le système d'exploitation. Il ne peut pas y avoir d'accès direct à la mémoire. L'accès au disque dur est réglementé dans une applet. Les applets fonctionnant sur le Web sont soumises aux restrictions suivantes dans la version 1.0 de Java : • aucun programme ne peut ouvrir, lire, écrire ou effacer un fichier sur le système de l'utilisateur • aucun programme ne peut lancer un autre programme sur le système de l'utilisateur • toute fenêtre créée par le programme est clairement identifiée comme étant une fenêtre Java, ce qui interdit par exemple la création d'une fausse fenêtre demandant un mot de passe • les programmes ne peuvent pas se connecter à d'autres sites Web que celui dont ils proviennent.
Java est économe	le pseudo code a une taille relativement petite car les bibliothèques de classes requises ne sont liées qu'à l'exécution.
Java est multitâche	il permet l'utilisation de threads qui sont des unités d'exécution isolées. La JVM, elle même, utilise plusieurs threads.

Il existe 2 types de programmes avec la version standard de Java : les applets et les applications. Une application autonome (stand alone program) est une application qui s'exécute sous le contrôle direct du système d'exploitation. Une applet est une application qui est chargée par un navigateur et qui est exécutée sous le contrôle d'un plug in de ce dernier.

Les principales différences entre une applet et une application sont :

- les applets n'ont pas de méthode main() : la méthode main() est appelée par la machine virtuelle pour exécuter une application.
- les applets ne peuvent pas être testées avec l'interpréteur mais doivent être intégrées à une page HTML, elle même visualisée avec un navigateur disposant d'un plug in sachant gérer les applets Java, ou testées avec l'applet viewer.

1.2. Un bref historique de Java

Les principaux événements de la vie de Java sont les suivants :

Année	Evénements
1995	mai : premier lancement commercial
1996	janvier : JDK 1.0.1
1996	septembre : lancement du JDC
1997	février : JDK 1.1
1998	décembre : lancement de J2SE 1.2 et du JCP
1999	décembre : lancement J2EE
2000	mai : J2SE 1.3
2002	février : J2SE 1.4
2004	septembre : J2SE 5.0
2006	mai : Java EE 5 décembre : Java SE 6.0

2008	décembre : Java FX 1.0
2009	décembre : Java EE 6
2010	janvier : rachat de Sun par Oracle

1.3. Les différentes éditions et versions de Java

Sun fourni gratuitement un ensemble d'outils et d'API pour permettre le développement de programmes avec Java. Ce kit, nommé JDK, est librement téléchargeable sur le site web de Sun http://java.sun.com ou par FTP ftp://java.sun.com/pub/

Le JRE (Java Runtime Environment) contient uniquement l'environnement d'exécution de programmes Java. Le JDK contient lui même le JRE. Le JRE seul doit être installé sur les machines où des applications Java doivent être exécutées.

Depuis sa version 1.2, Java a été renommé Java 2. Les numéros de version 1.2 et 2 désignent donc la même version. Le JDK a été renommé J2SDK (Java 2 Software Development Kit) mais la dénomination JDK reste encore largement utilisée, à tel point que la dénomination JDK est reprise dans la version 5.0. Le JRE a été renommé J2RE (Java 2 Runtime Environment).

Sun définit trois plateformes d'exécution (ou éditions) pour Java pour des cibles distinctes selon les besoins des applications à développer :

- Java Standard Edition (J2SE / Java SE) : environnement d'exécution et ensemble complet d'API pour des applications de type desktop. Cette plate-forme sert de base en tout ou partie aux autres plateformes
- Java Enterprise Edition (J2EE / Java EE) : environnement d'exécution reposant intégralement sur Java SE pour le développement d'applications d'entreprises
- Java Micro Edition (J2ME / Java ME) : environnement d'exécution et API pour le développement d'applications sur appareils mobiles et embarqués dont les capacités ne permettent pas la mise en oeuvre de Java SE

La séparation en trois plateformes permet au développeur de mieux cibler l'environnement d'exécution et de faire évoluer les plateformes de façon plus indépendante.

Avec différentes éditions, les types d'applications qui peuvent être développées en Java sont nombreux et variés :

- Applications desktop
- Applications web : servlets/JSP, portlets, applets
- Applications pour appareil mobile (MIDP) : midlets
- Applications pour appareil embarqué (CDC) : Xlets
- Applications pour carte à puce (Javacard) : applets Javacard
- Applications temps réel

Sun fourni le JDK, à partir de la version 1.2, sous les plate-formes Windows, Solaris et Linux.

La version 1.3 de Java est désignée sous le nom Java 2 version 1.3.

La version 1.5 de Java est désignée officiellement sous le nom J2SE version 5.0.

La version 6 de Java est désignée officiellement sous le nom Java SE version 6.

La documentation au format HTML des API de Java est fournie séparément. Malgré sa taille imposante, cette documentation est indispensable pour obtenir des informations complètes sur toutes les classes fournies. Le tableau ci dessous résume la taille des différents composants selon leur version pour la plate-forme Windows.

	Version 1.0	Version 1.1	Version 1.2	Version 1.3	Version 1.4	Version 5.0	Version 6
JDK compressé		8,6 Mo	20 Mo	30 Mo	47 Mo	44 Mo	73 Mo
JDK installé				53 Mo	59 Mo		

JRE compressé			12 Mo	7 Mo		14 Mo	15,5 Mo
JRE installé				35 Mo	40 Mo		
Documentation compressée			16 Mo	21 Mo	30 Mo	43,5 Mo	56 Mo
Documentation décompressée			83 Mo	106 Mo	144 Mo	223 Mo	

1.3.1. Les évolutions des plates-formes Java

Les technologies Java évoluent au travers du JCP (Java Community Process). Le JCP est une organisation communautaire ouverte qui utilise des processus définis pour définir ou réviser les spécifications des technologies Java.

Les membres du JCP sont des personnes individuelles ou membre d'organisations communautaires ou de sociétés commerciales qui tendent à mettre en adéquation la technologie Java avec les besoins du marché.

Bien que le JCP soit une organisation communautaire ouverte, Sun Microsystems est le détenteur des marques déposées autour de la technologie Java et l'autorité suprême concernant les plates-formes Java.

Des membres du JCP qui souhaitent enrichir la plate-forme Java doivent faire une proposition formalisée sous la forme d'une JSR (Java Specification Request). Chaque JSR suit un processus qui définit son cycle de vie autour de plusieurs étapes clés : drafts, review et approval.

Chaque JSR est sous la responsabilité d'un leader et traitée par un groupe d'experts.

Il est possible de souscrire à la liste de diffusion du JCP à l'url : http://jcp.org/participation/mail/

Cette liste de diffusion permet d'être informé sur les évolutions des JSR et des procédures du JCP et de participer à des revues publiques ou fournir les commentaires

Le site du JCP propose une liste des JSR par plates-formes ou technologies :

- Java SE : http://jcp.org/en/jsr/tech?listBy=2&listByType=platform
- Java EE : http://jcp.org/en/jsr/tech?listBy=3&listByType=platform
- Java ME : http://jcp.org/en/jsr/tech?listBy=1&listByType=platform
- OSS : http://jcp.org/en/jsr/tech?listBy=3&listByType=tech
- JAIN : http://jcp.org/en/jsr/tech?listBy=2&listByType=tech
- XML : http://jcp.org/en/jsr/tech?listBy=1&listByType=tech

Une fois validée, chaque JSR doit proposer une spécification, une implémentation de référence (Reference Implementation) et un technology compatibility kit (TCK).

1.3.2. Les différentes versions de Java

Chaque version de la plate-forme Java possède un numéro de version et un nom de projet.

A partir de la version 5, la plate-forme possède deux numéros de version :

- Un numéro de version interne : exemple 1.5.0
- Un numéro de version externe : exemple 5.0

Le nom de projet des versions majeures concerne des oiseaux ou des mammifères.

Le nom de projet des versions mineures concerne des insectes.

Version	Nom du projet	Date de diffusion
JDK 1.0	Oak	Mai 1995

JDK 1.1		Février 1997
JDK 1.1.4	Sparkler	Septembre 1997
JDK 1.1.5	Pumpkin	Décembre 1997
JDK 1.1.6	Abigail	Avril 1998
JDK 1.1.7	Brutus	Septembre 1998
JDK 1.1.8	Chelsea	Avril 1999
J2SE 1.2	Playground	Décembre 1998
J2SE 1.2.1		Mars 1999
J2SE 1.2.2	Cricket	Juillet 1999
J2SE 1.3	Kestrel	Mai 2000
J2SE 1.3.1	Ladybird	Mai 2001
J2SE 1.4.0	Merlin	Février 2002
J2SE 1.4.1	Hopper	Septembre 2002
J2SE 1.4.2	Mantis	Juin, 2003
J2SE 5.0 (1.5)	Tiger	Septembre 2004
Java SE 6.0 (1.6)	Mustang	Décembre 2006
Java SE 7.0 (1.7)	Dolphin	Fin 2010

1.3.3. Java 1.0

Cette première version est lancée officiellement en mai 1995.

Elle se compose de 8 packages :

- java.lang : classes de bases
- java.util : utilitaires
- java.applet : applets
- java.awt.* : interface graphique portable
- java.io : gestion des entrées/sorties grâce aux flux
- java.net : gestion des communications à travers le réseau

1.3.4. Java 1.1

Cette version du JDK est annoncée officiellement en mars 1997. Elle apporte de nombreuses améliorations et d'importantes fonctionnalités nouvelles dont :

- les Java beans
- les fichiers JAR
- RMI pour les objets distribués
- la sérialisation
- l'introspection
- JDBC pour l'accès aux données
- les classes internes
- l'internationalisation
- un nouveau modèle de sécurité permettant notamment de signer les applets
- un nouveau modèle de gestion des événements
- JNI pour l'appel de méthodes natives
- ...

1.3.5. Java 1.2 (nom de code Playground)

Cette version du JDK est lancée fin 1998. Elle apporte de nombreuses améliorations et d'importantes fonctionnalités nouvelles dont :

- un nouveau modèle de sécurité basé sur les policy
- les JFC sont incluses dans le JDK (Swing, Java2D, accessibility, drag & drop ...)
- JDBC 2.0
- les collections
- support de CORBA
- un compilateur JIT est inclus dans le JDK
- de nouveaux formats audio sont supportés
- ...

Java 2 se décline en 3 éditions différentes qui regroupent des APIs par domaine d'application :

- Java 2 Micro Edition (J2ME) : contient le nécessaire pour développer des applications capable de fonctionner dans des environnements limités tels que les assistants personnels (PDA), les téléphones portables ou les systèmes de navigation embarqués
- Java 2 Standard Edition (J2SE) : contient le nécessaire pour développer des applications et des applets. Cette édition reprend le JDK 1.0 et 1.1.
- Java 2 Enterprise Edition (J2EE) : contient un ensemble de plusieurs API permettant le développement d'applications destinées aux entreprises tel que JDBC pour l'accès aux bases de données, EJB pour développer des composants orientés métiers, Servlet / JSP pour générer des pages HTML dynamiques, ... Cette édition nécessite le J2SE pour fonctionner.

Le but de ces trois éditions est de proposer une solution reposant sur Java quelque soit le type de développement à réaliser.

1.3.6. J2SE 1.3 (nom de code Kestrel)

Cette version du JDK est lancée en mai 2000. Elle apporte de nombreuses améliorations notamment sur les performances et des fonctionnalités nouvelles dont :

- JNDI est inclus dans le JDK
- hotspot est inclus dans la JVM
- ...

La rapidité d'exécution a été grandement améliorée dans cette version.

1.3.7. J2SE 1.4 (nom de code Merlin)

Cette version du JDK, lancée début 2002, est issue des travaux de la JSR 59. Elle apporte de nombreuses améliorations notamment sur les performances et des fonctionnalités nouvelles dont :

- JAXP est inclus dans le JDK pour le support de XML
- JDBC version 3.0
- new I/O API pour compléter la gestion des entrée/sortie
- logging API pour la gestion des logs applicatives
- une API pour utiliser les expressions régulières
- une api pour gérer les préférences utilisateurs
- JAAS est inclus dans le JDK pour l'authentification
- un ensemble d'API pour utiliser la cryptographie
- les exceptions chainées
- l'outil Java WebStart

* ...

Cette version ajoute un nouveau mot clé au langage pour utiliser les assertions : assert.

1.3.8. J2SE 5.0 (nom de code Tiger)

La version 1.5 du J2SE est spécifiée par le JCP sous la JSR 176. Elle intègre un certain nombre de JSR dans le but de simplifier les développements en Java.

Ces évolutions sont réparties dans une quinzaine de JSR qui seront intégrées dans la version 1.5 de Java.

JSR-003	JMX Management API
JSR-013	Decimal Arithmetic
JSR-014	Generic Types
JSR-028	SASL
JSR-114	JDBC API Rowsets
JSR-133	New Memory Model and thread
JSR-163	Profiling API
JSR-166	Concurrency Utilities
JSR-174	Monitoring and Management for the JVM
JSR-175	Metadata facility
JSR-199	Compiler APIs
JSR-200	Network Transfer Format for Java Archives
JSR-201	Four Language Updates
JSR-204	Unicode Surrogates
JSR-206	JAXP 1.3

La version 1.5 de Java est désignée officiellement sous le nom J2SE version 5.0.

La technologie Pack200 permet de compresser les fichiers .jar pour obtenir des gains pouvant atteindre 60%.

1.3.9. Java SE 6 (nom de code Mustang)

Cette version est spécifiée par le JCP sous la JSR 270 et développée sous le nom de code Mustang.

Elle intègre un changement de dénomination et de numérotation : la plate-forme J2SE est renommée en Java SE, SE signifiant toujours Standard Edition.

Cette version inclue les JSR :

JSR 105	XML Digital Signature APIs
JSR 173	Streaming API for XML
JSR 181	Web Services Metadata for Java Platform
JSR 199	Java Compiler API

JSR 202	Java Class File Specification Update
JSR 221	JDBC 4.0 API Specification
JSR 222	Java Architecture for XML Binding (JAXB) 2.0
JSR 223	Scripting for the Java Platform
JSR 224	Java API for XML-Based Web Services (JAX-WS) 2.0
JSR 250	Common Annotations for the Java Platform
JSR 269	Pluggable Annotation Processing API

Elle apporte plusieurs améliorations :

L'amélioration du support XML

Java 6.0 s'est enrichie avec de nombreuses nouvelles fonctionnalités concernant XML :

- JAXP 1.4
- JSR 173 Streaming API for XML (JSR 173)
- JSR 222 Java Architecture for XML Binding (JAXB) 2.0 : évolution de JAXB qui utilise maintenant les schémas XML
- Le support du type de données XML de la norme SQL 2003 dans JDBC

JDBC 4.0

Cette nouvelle version de l'API JDBC est le fruit des travaux de la JSR 221. Elle apporte de nombreuses évolutions :

- chargement automatique des pilotes JDBC
- Support du type SQL ROWID et XML
- Amélioration du support des champs BLOB et CLOB
- L'exception SQLTransaction possède deux classes filles SQLTransientException et SQLNonTransientException

Le support des services web

Les services web font leur apparition dans la version SE de Java : précédemment ils n'étaient intégrés que dans la version EE. Plusieurs JSR sont ajoutés pour supporter les services web dans la plate-forme :

- JSR 224 Java API for XML based Web Services (JAX-WS) 2.0 : facilite le développement de services web et de proposer un support des standards SOAP 1.2, WSDL 2.0 et WS-I Basic Profile 1.1
- JSR 181 Web Services MetaData : défini un ensemble d'annotations pour faciliter le développement de services web
- JSR 105 XML Digital Signature API : la package javax.xmlcrypto contient une API qui implémente la spécification Digital Signature du W3C. Ceci permet de proposer une solution pour sécuriser les services web

Le support des moteurs de scripts

L'API Scripting propose un standard pour l'utilisation d'outils de scripting. Cette API a été développée sous la JSR 223. La plate-forme intègre Rhino un moteur de scripting Javascript

L'amélioration de l'intégration dans le système d'exploitation sous jacent

- La classe java.awt.SystemTray permet d'interagir avec la barre d'outils système (System Tray)

- La classe java.awt.Desktop qui permet des interactions avec le système d'exploitation (exécution de fonctionnalités de base sur des documents selon leur type)
- Nouveaux modes pour gérer la modalité d'une fenêtre
- Amélioration dans Swing
- Amélioration des look and feel Windows et GTK
- la classe javax.awt.SplashScreen qui propose un support des splashscreens
- Ajout du layout javax.swing.GroupLayout
- Filtres et tris des données dans le composant Jtable
- La classe SwingWorker qui facilite la mise en oeuvre de threads dans Swing
- Utilisation du double buffering pour l'affichage

L'améliorations dans l'API Collection

- 5 nouvelles interfaces : Deque (queue à double sens), BlockingDeque, NavigableSet (étend SortedSet), NavigableMap (étend SortedMap) et ConcurrentNavigableMap
- ArrayDeque : implémentation de Deque utilisant un tableau
- ConcurrentSkipListSet : implémentation de NavigableSet
- ConcurrentSkipListMap : implémentation de ConcurrentNavigableMap
- LinkedBlockingDeque : implémentation de BlockingDeque

L'améliorations dans l'API IO

- Modification des attributs d'un fichier grâce aux méthodes setReadable(), setWritable() et setExecutable de la classe File

Java Compiler API

Cette API est le résultat des travaux de la JSR 199 et a pour but de proposer une utilisation directe du compilateur Java. Cette API est utilisable à partir du package javax.tools

Pluggable Annotation-Processing API

Cette API est le résultat des travaux de la JSR 269 et permet un traitement des annotations à la compilation. Cette API est utilisable à partir du package javax.annotation.processing

Common Annotations

Cette API est le résultat des travaux de la JSR 250 et définit plusieurs nouvelles annotations standards.
@javax.annotation.Generated : permet de marque une classe, une méthode ou un champ comme étant généré par un outil
@javax.annotation.PostConstruct : méthode exécutée après la fin de l'injection de dépendance
@javax.annotation.PreDestroy : méthode de type callback appelée juste avant d'être supprimé par le conteneur
@javax.annotation.Resource : permet de déclarer une référence vers une ressource
@javax.annotation.Resources : conteneur pour la déclaration de plusieurs ressources

Java Class File Specification

Issue des travaux de la JSR 202, cette spécification fait évoluer le format du fichier .class résultant de la compilation.

La vérification d'un fichier .class exécute un algorithme complexe et coûteux en ressources et en temps d'exécution pour valider un fichier .class.

La JSR 202, reprend une technique développée pour le profile CLDC de J2ME nommée split vérification qui décompose la vérification d'un fichier .class en deux étapes :

- la première étape réalisée lors de la création du fichier .class ajoute des attributs qui seront utilisés par la seconde étape
- la seconde étape est réalisée à l'exécution en utilisant les attributs

Le temps de chargement du fichier .class est ainsi réduit.

Le Framework JavaBeans Activation

Le Framework JavaBeans Activation a été intégré en standard dans la plate-forme Java SE 6. Ce framework historiquement fourni séparément permet de gérer les types mimes et était généralement utilisé avec l'API JavaMail. Ce framework permet d'associer des actions à des types mimes.

La liste des nouveaux packages de Java 6 comprend :

java.text.spi	
java.util.spi	
javax.activation	Activation Framework
javax.annotation	Traitement des annotations
javax.jws	Support des services web
javax.jws.soap	support SOAP
javax.lang.model.*	
javax.script	Support des moteurs de scripting
javax.tools	Accès à certains outils notamment le compilateur
javax.xml.bind.*	JAXB
javax.xml.crypto.*	Cryptographie avec XML
javax.xml.soap	Support des messages SOAP
javax.xml.stream.*	API Stax
javax.xml.ws.*	API JAX-WS

Une base de données nommée JavaDB est ajoutée au JDK 6.0 : c'est une version de la base de données Apache Derby.

1.3.10. Java 6 update

En attendant la version 7, Sun a proposé plusieurs mises à jour de la plate-forme Java SE. Ces mises à jour concernent :

- des corrections de bugs et ou de sécurité
- des évolutions ou des ajouts dans les outils du JDK et du JRE

Deux de ces mises à jour sont particulièrement importantes : update 10 et 14.

1.3.10.1. Java 6 update 1

Cette mise à jour contient des corrections de bugs.

La liste complète est consultable à l'url : http://java.sun.com/javase/6/webnotes/6u1.html

1.3.10.2. Java 6 update 2

Cette mise à jour contient des corrections de bugs et une nouvelle version de la base de données embarquée Java DB.

La liste complète est consultable à l'url : http://java.sun.com/javase/6/webnotes/6u2.html

1.3.10.3. Java 6 update 3

Cette mise à jour contient des corrections de bugs.

La liste complète est consultable à l'url : http://java.sun.com/javase/6/webnotes/6u3.html

1.3.10.4. Java 6 update 4

Cette mise à jour contient des corrections de bugs et la version 10.3 de Java DB.

La liste complète est consultable à l'url : http://java.sun.com/javase/6/webnotes/6u4.html

1.3.10.5. Java 6 update 5

Cette mise à jour contient des corrections de bugs et la possibilité d'enregistrer le JDK.

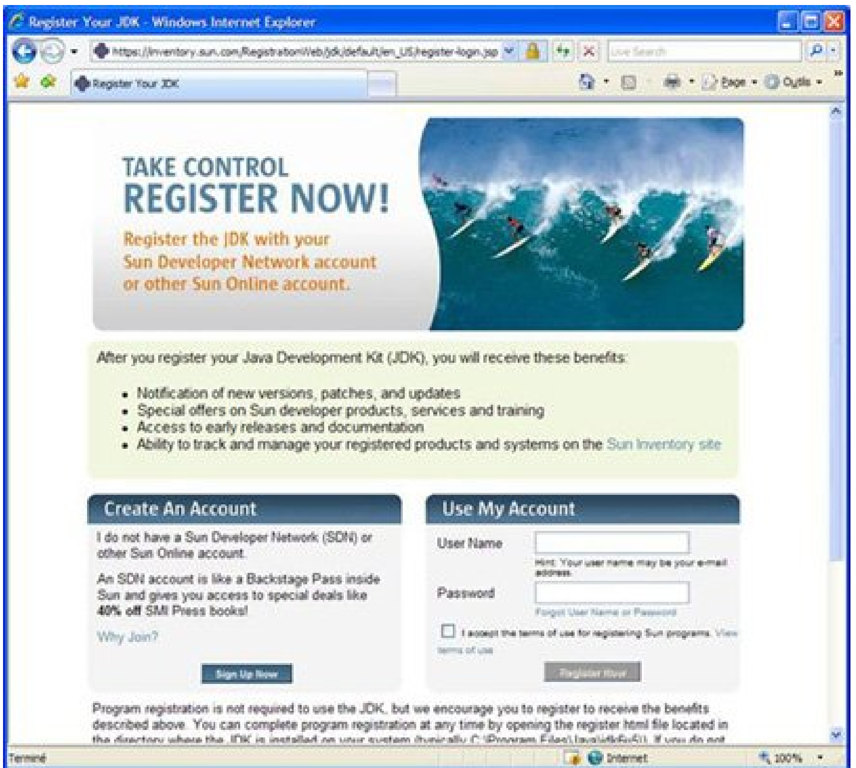
La liste complète est consultable à l'url : http://java.sun.com/javase/6/webnotes/6u5.html

Depuis la version 6u5 de Java, le programme d'installation du JDK propose à la fin la possibilité d'enregistrer le JDK.

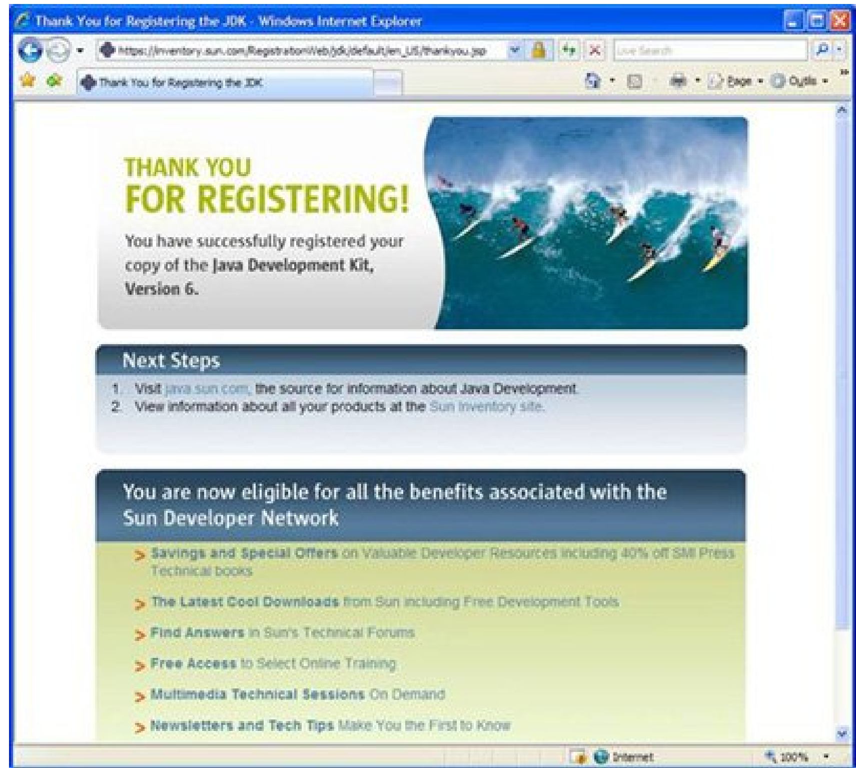
Il suffit de cliquer sur le bouton « Product Registration Information » pour obtenir des informations sur le processus d'enregistrement du produit.

Lors du clic sur le bouton « Finish », le processus d'enregistrement collecte les informations sur le JDK installé et sur le système hôte. Ces informations sont envoyées via une connection http sécurisée sur le serveur Sun Connection.

Le navigateur s'ouvre sur la page d'enregistrement du JDK.



Il faut utiliser son compte SDN (Sun Developer Network) pour se logguer et afficher la page « Thank You ».



Il est possible d'enregistrer son JDK en ouvrant la page register.html situé dans le répertoire d'installation du JDK.

En plus du JDK, plusieurs autres produits de Sun Connection peuvent être enregistrés comme GlassFish, Netbeans, ...

Sun Connection propose un service gratuit nommé Inventory Channel qui permet de gérer ses produits enregistrés.

1.3.10.6. Java 6 update 6

Cette mise à jour contient des corrections de bugs.

La liste complète est consultable à l'url : http://java.sun.com/javase/6/webnotes/6u6.html

1.3.10.7. Java 6 update 7

Cette mise à jour contient des corrections de bugs et l'outil Java Visual VM.

La liste complète est consultable à l'url : http://java.sun.com/javase/6/webnotes/6u7.html

Le numéro de version interne complet est build 1.6.0_07-b06. Le numéro de version externe est 6u7.

Exemple :

```
C:\>java -version
java version "1.6.0_07"
Java(TM) SE Runtime Environment (build 1.6.0_07-b06)
Java HotSpot(TM) Client VM (build 10.0-b23, mixed mode, sharing)
```

1.3.10.8. Java 6 update 10

Cette mise à jour qui a porté le nom Java Update N ou Consumer JRE est très importante car elle apporte de grandes évolutions notamment pour le support des applications de type RIA.

Elle contient :

- Un nouveau plug-in pour l'exécution des applets dans les navigateurs
- Nimbus, un nouveau L&F pour Swing
- Java kernel
- Java Quick Starter (JQS)
- des corrections de bugs
- la version 10.4 Java DB
- support par défaut de Direct3D 9 sous Windows.

La liste complète est consultable à l'url : http://java.sun.com/javase/6/webnotes/6u10.html

Le plug-in pour les navigateurs a été entièrement réécrit notamment pour permettre une meilleure exécution des applets, des applications Java Web Start et des applications RIA en Java FX.

1.3.10.9. Java 6 update 11

Cette mise à jour ne contient que des corrections de bugs et des patchs de sécurité.

La liste complète est consultable à l'url : http://java.sun.com/javase/6/webnotes/6u11.html

La version externe est Java 6u11, le numéro de build est 1.6.0_11-b03.

1.3.10.10. Java 6 update 12

Cette mise à jour contient

- Support de Windows Server 2008
- Amélioration des performances graphiques et de démarrage des applications Java Web Start
- des corrections de bugs.

La liste complète est consultable à l'url : http://java.sun.com/javase/6/webnotes/6u12.html

La version externe est Java 6u12, le numéro de build est 1.6.0_12-b04.

1.3.10.11. Java 6 update 13

Cette mise à jour contient des corrections de bugs.

La liste complète est consultable à l'url : http://java.sun.com/javase/6/webnotes/6u13.html

La version externe est Java 6u13, le numéro de build est 1.6.0_13-b03.

L'installeur propose par défaut l'installation de la barre d'outils de Yahoo.

1.3.10.12. Java 6 update 14

Cette mise à jour contient

* La version 14 de la JVM HotSpot
* G1, le nouveau ramasse miette
* Les versions 2.1.6 de JAX-WS et 2.1.10 de JAX-B
* La version 10.4.2.1 de Java DB
* Des mises à jour dans Visual VM
* Blacklist des jars signés qui contiennent une faille de sécurité
* des corrections de bugs.

La liste complète est consultable à l'url : http://java.sun.com/javase/6/webnotes/6u14.html

La version externe est Java 6u14, le numéro de build est 1.6.0_14-b08.

1.3.10.13. Java 6 update 15

Cette mise à jour ne contient que des corrections de bugs et patchs de sécurité.

La liste complète est consultable à l'url : http://java.sun.com/javase/6/webnotes/6u15.html

La version externe est Java 6u15, le numéro de build est 1.6.0_15-b03.

1.3.10.14. Java 6 update 16

Cette mise à jour ne contient que des corrections de bugs.

La liste complète est consultable à l'url : http://java.sun.com/javase/6/webnotes/6u16.html

La version externe est Java 6u16, le numéro de build est 1.6.0_16-b01.

1.3.11. Les futures versions de Java

La prochaine version de Java (la version 7) porte le nom de code Dolphin. Cette version sera la première distribuée sous la licence GPL 2. Elle devrait être diffusée en 2010.

Le site officiel de cette version est à l'url https://jdk7.dev.java.net/

Le site http://java.net/ est le portail des projets open source en Java.

1.3.12. Le résumé des différentes versions

Au fur et à mesure des nouvelles versions de Java, le nombre de packages et de classes s'accroît :

	Java 1.0	Java 1.1	Java 1.2	J2SE 1.3	J2SE 1.4	J2SE 5.0	Java SE 6
Nombre de packages	8	23	59	76	135	166	202
Nombre de classes	201	503	1520	1840	2990	3280	3780

1.3.13. Les extensions du JDK

Sun fourni un certain nombre d'API supplémentaires qui ne sont pas initialement fournies en standard dans le JDK. Ces API sont intégrées au fur et à mesure de l'évolution de Java.

Extension	Description
JNDI	Java Naming and directory interface Cet API permet d'unifier l'accès à des ressources. Elle est intégrée à Java 1.3
Java mail	Cette API permet de gérer des emails. Elle est intégrée à la plateforme J2EE.
Java 3D	Cette API permet de mettre en oeuvre des graphismes en 3 dimensions
Java Media	Cette API permet d'utiliser des composants multimédia
Java Servlets	Cette API permet de créer des servlets (composants serveurs). Elle est intégrée à la plateforme J2EE.
Java Help	Cette API permet de créer des aides en ligne pour les applications
Jini	Cette API permet d'utiliser Java avec des appareils qui ne sont pas des ordinateurs
JAXP	Cette API permet le parsing et le traitement de document XML. Elle est intégré à Java 1.4

Cette liste n'est pas exhaustive.

1.4. Un rapide tour d'horizon des API et de quelques outils

La communauté Java est très productive car elle regroupe :

- Sun, le fondateur de Java
- le JCP (Java Community Process) : c'est le processus de traitement des évolutions de Java dirigé par Sun. Chaque évolution est traitée dans une JSR (Java Specification Request) par un groupe de travail constitué de différents acteurs du monde Java
- des acteurs commerciaux dont tous les plus grands acteurs du monde informatique excepté Microsoft
- la communauté libre qui produit un très grand nombre d'API et d'outils pour Java

Ainsi l'ensemble des API et des outils utilisables est énorme et évolue très rapidement. Les tableaux ci dessous tentent de recenser les principaux par thème.

J2SE 1.4			
Java Bean	RMI	IO	Applet
Reflexion	Collection	Logging	AWT
Net (réseau)	Preferences	Security	JFC
Internationalisation	Exp régulière		Swing

Les outils de Sun			
Jar	Javadoc	Java Web Start	JWSDK

Les outils libres (les plus connus)			
Jakarta Tomcat	Jakarta Ant	JBoss	Apache Axis
JUnit	Eclipse	NetBeans	Maven

JOnas			

Les autres API				
Données	Web	Entreprise	XML	Divers
JDBC	Servlets	Java Mail	JAXP	JAI
JDO	JSP	JNDI	SAX	JAAS
JPA	JSTL	EJB	DOM	JCA
	Jave Server Faces	JMS	JAXB	JCE
		JMX	Stax	Java Help
		JTA	Services Web	JMF
		RMI-IIOP	JAXM	JSSE
		Java IDL	JAXR	Java speech
		JINI	JAX-RPC	Java 3D
		JXTA	SAAJ	
			JAX-WS	

Les API de la communauté open source				
Données	Web	Entreprise	XML	Divers
OJB	Jakarta Struts	Spring	Apache Xerces	Jakarta Log4j
Castor	Webmacro	Apache Axis	Apache Xalan	Jakarta regexp
Hibernate	Expresso	Seams	JDOM	
	Barracuda		DOM4J	
	Turbine			
	GWT			

1.5. Les différences entre Java et JavaScript

Il ne faut pas confondre Java et JavaScript. JavaScript est un langage développé par Netscape Communications.

La syntaxe des deux langages est très proche car elles dérivent toutes les deux du C++.

Il existe de nombreuses différences entre les deux langages :

	Java	Javascript
Auteur	Développé par Sun Microsystems	Développé par Netscape Communications
Format	Compilé sous forme de byte-code	Interprété
Stockage	Applet téléchargé comme un élément de la page web	Source inséré dans la page web

Utilisation	Utilisable pour développer tous les types d'applications	Utilisable uniquement pour "dynamiser" les pages web
Exécution	Exécuté dans la JVM du navigateur	Exécuté par le navigateur
POO	Orienté objets	Manipule des objets mais ne permet pas d'en définir
Typage	Fortement typé	Pas de contrôle de type
Complexité du code	Code relativement complexe	Code simple

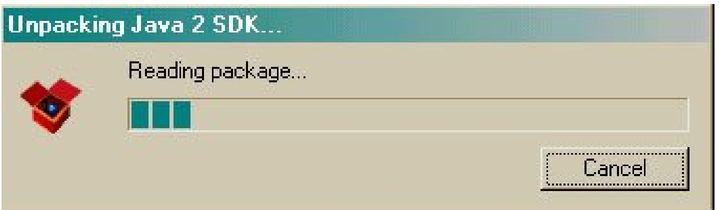
1.6. L'installation du JDK

Le JDK et la documentation sont librement téléchargeables sur le site web de Sun : http://java.sun.com

1.6.1. L'installation de la version 1.3 DU JDK de Sun sous Windows 9x

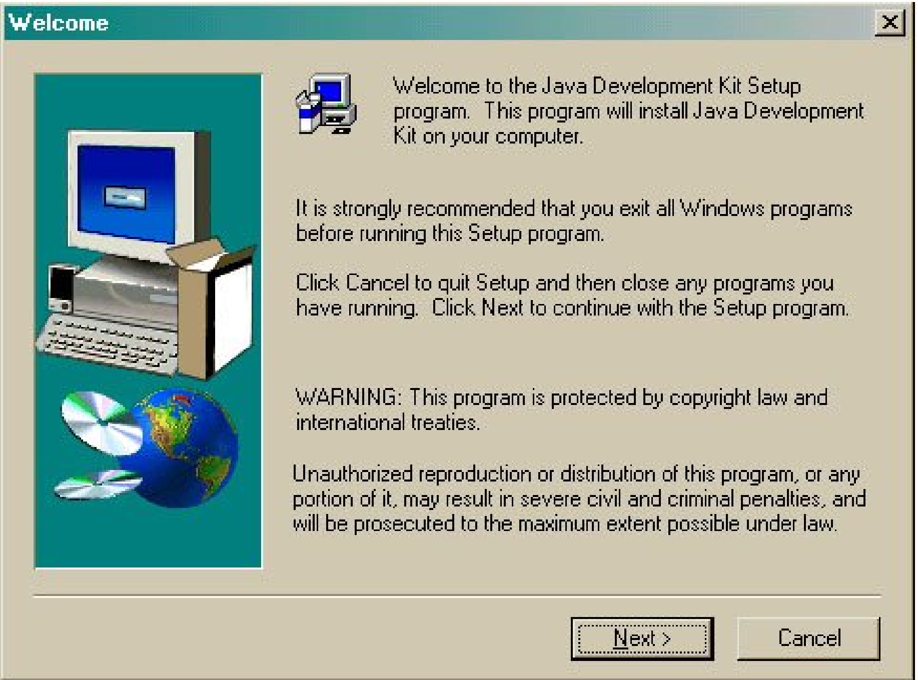
Pour installer le JDK 1.3 sous Windows 9x, il suffit de télécharger et d'exécuter le programme : j2sdk1_3_0-win.exe

Le programme commence par désarchiver les composants.

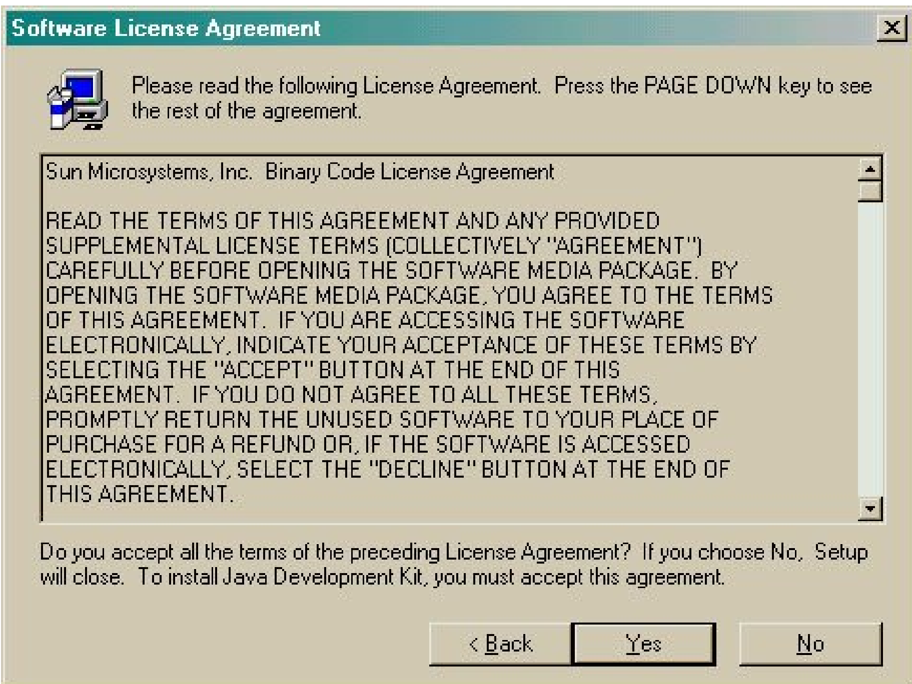


Le programme utilise InstallShield pour guider et réaliser l'installation.

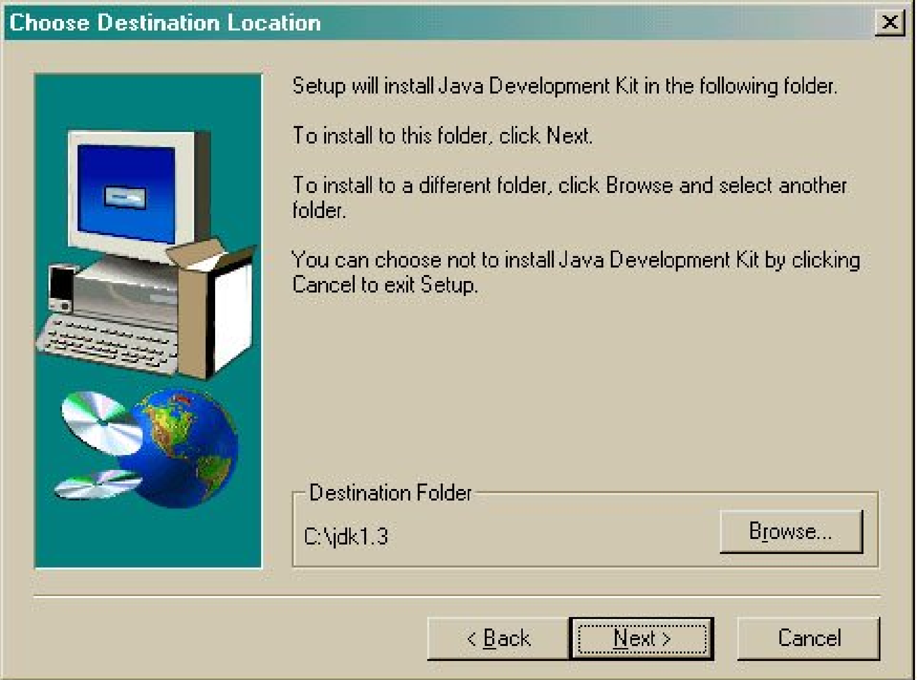
L'installation vous souhaite la bienvenue et vous donne quelques informations d'usage.



L'installation vous demande ensuite de lire et d'approuver les termes de la licence d'utilisation.

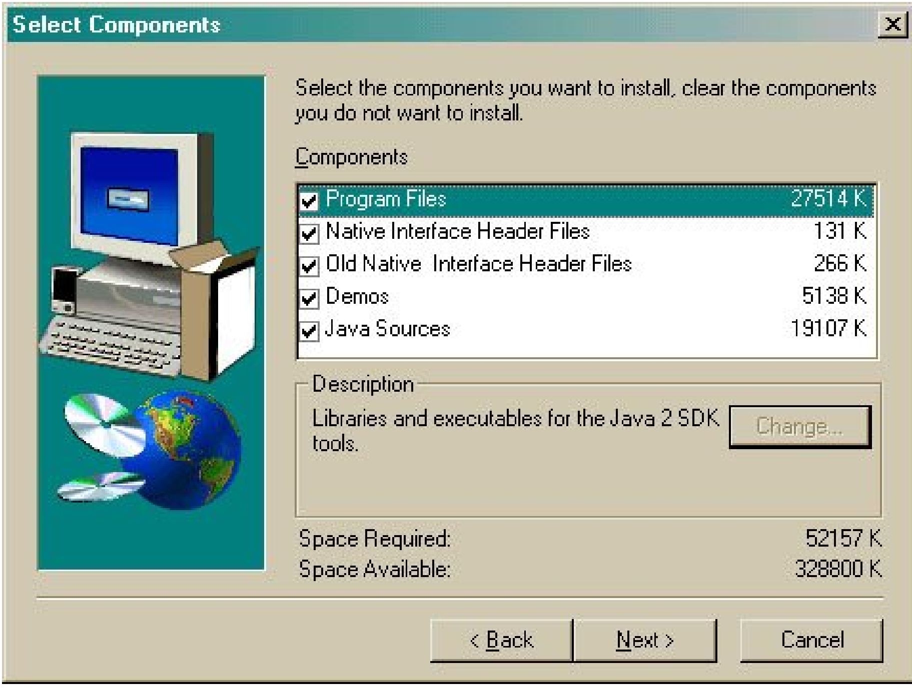


L'installation vous demande le répertoire dans lequel le JDK va être installé. Le répertoire proposé par défaut est pertinent car il est simple.



L'installation vous demande les composants à installer :

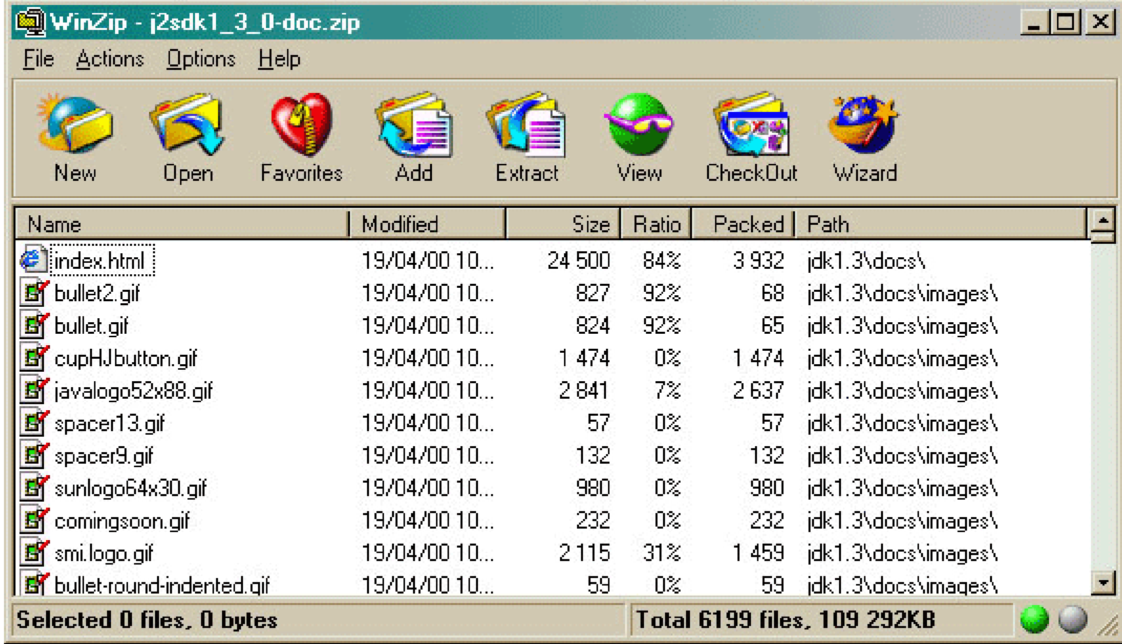
- Program Files est obligatoire pour une première installation
- Les interfaces natives ne sont utiles que pour réaliser des appels de code natif dans les programmes Java
- Les démos sont utiles car ils fournissent quelques exemples
- les sources contiennent les sources de la plupart des classes Java écrites en Java. Attention à l'espace disque nécessaire à cet élément



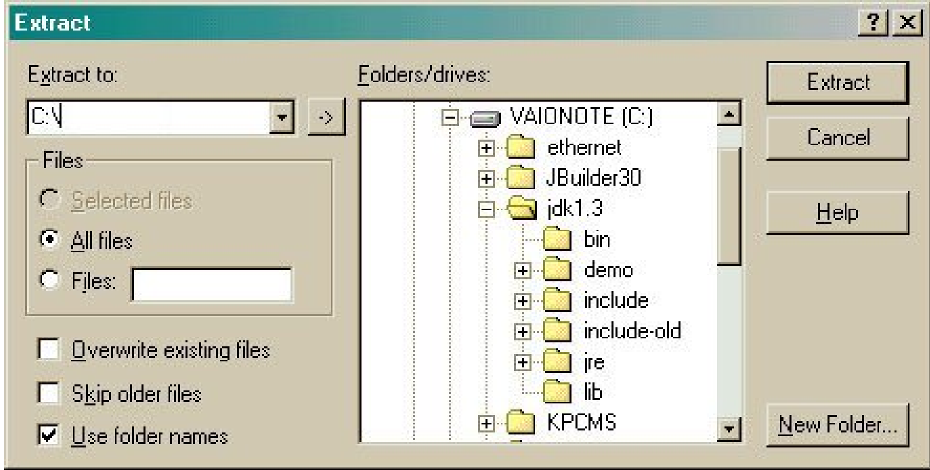
L'installation se poursuit par la copie des fichiers et la configuration du JRE.

1.6.2. L'installation de la documentation de Java 1.3 sous Windows

L'archive contient la documentation sous forme d'arborescence dont la racine est jdk1.3\docs.



Si le répertoire par défaut a été utilisé lors de l'installation, il suffit de décompresser l'archive à la racine du disque C:\.

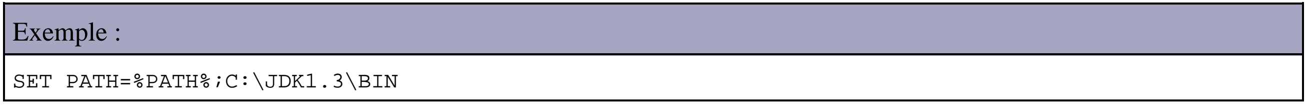


Il peut être pratique de désarchiver le fichier dans un sous répertoire, ce qui permet de réunir plusieurs versions de la documentation.

1.6.3. La configuration des variables système sous Windows 9x

Pour un bon fonctionnement du JDK, il est recommandé de paramétrer correctement deux variables systèmes : la variable PATH qui définit les chemins de recherche des exécutables et la variable CLASSPATH qui définit les chemins de rechercher des classes et bibliothèques Java.

Pour configurer la variable PATH, il suffit d'ajouter à la fin du fichier autoexec.bat :


Exemple :

```
SET PATH=%PATH%;C:\JDK1.3\BIN
```

Attention : si une version antérieure du JDK était déjà présente, la variable PATH doit déjà contenir un chemin vers les utilitaires du JDK. Il faut alors modifier ce chemin sinon c'est l'ancienne version qui sera utilisée. Pour vérifier la version du JDK utilisée, il suffit de saisir la commande java -version dans une fenêtre DOS.

La variable CLASSPATH est aussi définie dans le fichier autoexec.bat. Il suffit d'ajouter une ligne ou de modifier la ligne existante définissant cette variable.

Exemple :

```
SET CLASSPATH=C:\JAVA\DEV;
```

Dans un environnement de développement, il est pratique d'ajouter le . qui désigne le répertoire courant dans le CLASSPATH surtout lorsque l'on n'utilise pas d'outils de type IDE. Attention toutefois, cette pratique est fortement déconseillée dans un environnement de production pour ne pas poser des problèmes de sécurité.

Il faudra ajouter par la suite les chemins d'accès aux différents packages requis par les développements afin de les faciliter.

Pour que ces modifications prennent effet dans le système, il faut redémarrer Windows ou exécuter ces deux instructions sur une ligne de commande DOS.

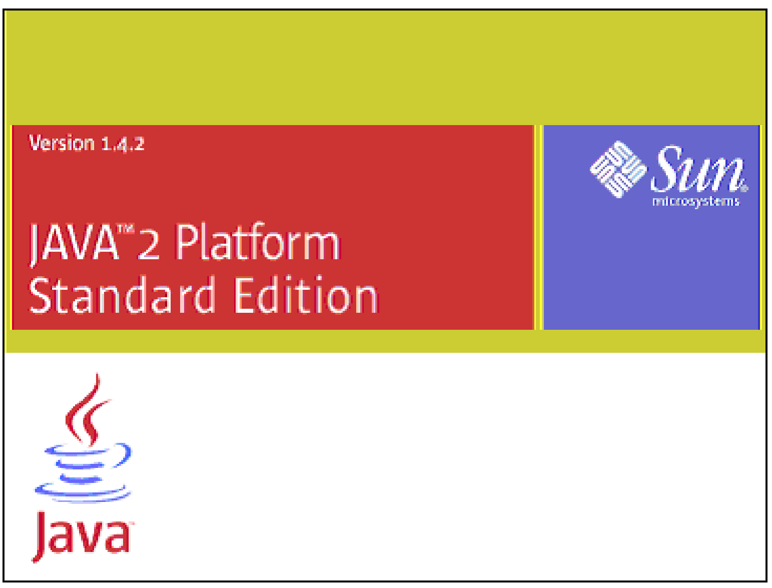
1.6.4. Les éléments du JDK 1.3 sous Windows

Le répertoire dans lequel a été installé le JDK contient plusieurs répertoires. Les répertoires donnés ci-après sont ceux utilisés en ayant gardé le répertoire par défaut lors de l'installation.

Répertoire	Contenu
C:\jdk1.3	Le répertoire d'installation contient deux fichiers intéressants : le fichier readme.html qui fourni quelques informations et des liens web et le fichier src.jar qui contient le source Java de nombreuses classes. Ce dernier fichier n'est présent que si l'option correspondante a été cochée lors de l'installation.
C:\jdk1.3\bin	Ce répertoire contient les exécutables : le compilateur javac, l'interpréteur java, le débuggeur jdb et d'une façon générale tous les outils du JDK.
C:\jdk1.3\demo	Ce répertoire n'est présent que si l'option nécessaire a été cochée lors de l'installation. Il contient des applications et des applets avec leur code source.
C:\jdk1.3\docs	Ce répertoire n'est présent que si la documentation a été décompressée.
C:\jdk1.3\include et C:\jdk1.3\include-old	Ces répertoires ne sont présents que si les options nécessaires ont été cochées lors de l'installation. Il contient des fichiers d'en-tête C (fichier avec l'extension .H) qui permettent de faire interagir du code Java avec du code natif
C:\jdk1.3\jre	Ce répertoire contient le JRE : il regroupe le nécessaire à l'exécution des applications notamment le fichier rt.jar qui regroupe les API. Depuis la version 1.3, le JRE contient deux machines virtuelles : la JVM classique et la JVM utilisant la technologie Hot spot. Cette dernière est bien plus rapide et c'est elle qui est utilisée par défaut. Les éléments qui composent le JRE sont séparés dans les répertoires bin et lib selon leur nature.
C:\jdk1.3\lib	Ce répertoire ne contient plus que quelques bibliothèques notamment le fichier tools.jar. Avec le JDK 1.1 ce répertoire contenait le fichier de la bibliothèque standard. Ce fichier est maintenant dans le répertoire JRE.

1.6.5. L'installation de la version 1.4.2 du JDK de Sun sous Windows

Télécharger sur le site java.sun.com et exécuter le fichier j2sdk-1_4_2_03-windows-i586-p.exe.



Un assistant permet de configurer l'installation au travers de plusieurs étapes :

* La page d'acceptation de la licence (« Licence agreement ») s'affiche
* Lire la licence et si vous l'acceptez, cliquer sur le bouton radio « I accept the terms in the licence agreement », puis cliquez sur le bouton « Next »
* La page de sélection des composants à installer (« Custom setup ») s'affiche, modifiez les composants à installer si nécessaire puis cliquez sur le bouton « Next »
* La page de sélection des plug in pour navigateur (« Browser registration ») permet de sélectionner les navigateurs pour lesquels le plug in Java sera installé, sélectionner ou non le ou les navigateurs détecté, puis cliquez sur le bouton « Install »
* L'installation s'opère en fonction des informations fournies précédemment
* La page de fin s'affiche, cliquez sur le bouton « Finish »

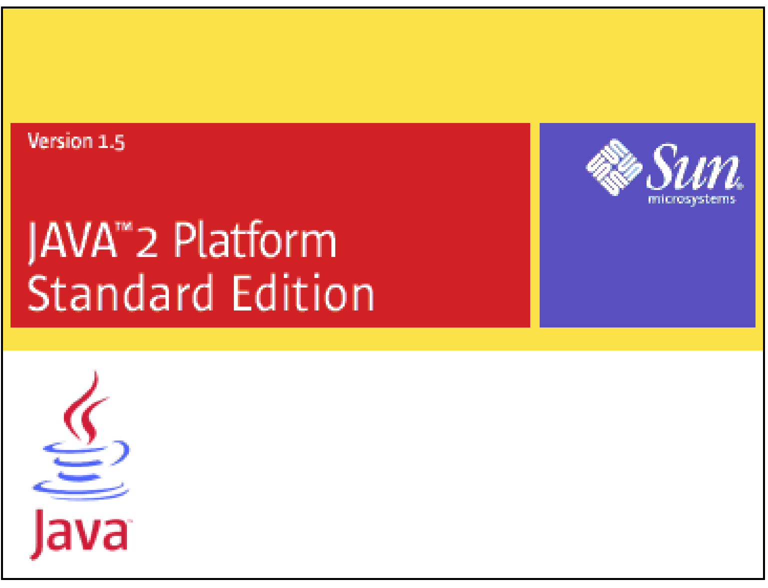
Même si ce n'est pas obligatoire pour fonctionner, il est particulièrement utile de configurer deux variables systèmes : PATH et CLASSPATH.

Dans la variable PATH, il est pratique de rajouter le chemin du répertoire bin du JDK installé pour éviter à chaque appel des commandes du JDK d'avoir à saisir leur chemin absolu.

Dans la variable CLASSPATH, il est pratique de rajouter les répertoires et les fichiers .jar qui peuvent être nécessaire lors des phases de compilation ou d'exécution, pour éviter d'avoir à les préciser à chaque fois.

1.6.6. L'installation de la version 1.5 du JDK de Sun sous Windows

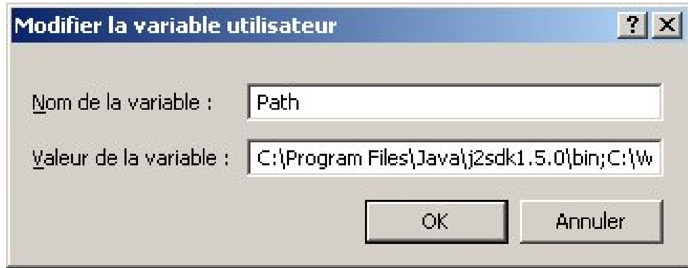
Il faut télécharger sur le site de Sun et exécuter le fichier j2sdk-1_5_0-windows-i586.exe



Un assistant guide l'utilisateur pour l'installation de l'outil.

- Sur la page « Licence Agreement », il faut lire la licence et si vous l'acceptez, cochez le bouton radio « I accept the terms in the licence agreement » et cliquez sur le bouton « Next »
- Sur la page « Custom Setup », il est possible de sélectionner/désélectionner les éléments à installer. Cliquez simplement sur le bouton « Next ».
- La page « Browser registration » permet de sélectionner les plug-ins des navigateurs qui seront installés. Cliquez sur le bouton « Install »
- Les fichiers sont copiés.
- La page « InstallShield Wizard Completed » s'affichage à la fin de l'installation. Cliquez sur « Finish ».

Pour faciliter l'utilisation des outils du J2SE SDK, il faut ajouter le chemin du répertoire bin contenant ces outils dans la variable Path du système.



Il est aussi utile de définir la variable d'environnement JAVA_HOME avec comme valeur le chemin d'installation du SDK.

1.6.7. Installation JDK 1.4.2 sous Linux Mandrake 10

La première chose est de décompresser le fichier téléchargé sur le site de Sun en exécutant le fichier dans un shell.

Exemple :

```
[java@localhost tmp]$ sh j2sdk-1_4_2_06-linux-i586-rpm.bin
              Sun Microsystems, Inc.
          Binary Code License Agreement
                   for the

JAVATM 2 SOFTWARE DEVELOPMENT KIT (J2SDK), STANDARD
EDITION, VERSION 1.4.2_X

SUN MICROSYSTEMS, INC. ("SUN") IS WILLING TO LICENSE THE
SOFTWARE IDENTIFIED BELOW TO YOU ONLY UPON THE CONDITION
```

```
THAT YOU ACCEPT ALL OF THE TERMS CONTAINED IN THIS BINARY
CODE LICENSE AGREEMENT AND SUPPLEMENTAL LICENSE TERMS
(COLLECTIVELY "AGREEMENT"). PLEASE READ THE AGREEMENT
...

Do you agree to the above license terms? [yes or no]
yes
Unpacking...
Checksumming...
0
0
Extracting...
UnZipSFX 5.40 of 28 November 1998, by Info-ZIP (Zip-Bugs@lists.wku.edu).
  inflating: j2sdk-1_4_2_06-linux-i586.rpm
Done.
[java@localhost tmp]$
```

La décompression créé un fichier j2sdk-1_4_2_06-linux-i586.rpm. Pour installer ce package, il est nécessaire d'être root sinon son installation est impossible.

Exemple :

```
[java@localhost eclipse3]$ rpm -ivh j2sdk-1_4_2_06-linux-i586.rpm
erreur: cannot open lock file ///var/lib/rpm/RPMLOCK in exclusive mode
erreur: impossible d'ouvrir la base de données Package dans /var/lib/rpm

[java@localhost eclipse3]$ su root
Password:
[root@localhost eclipse3]# rpm -ivh j2sdk-1_4_2_06-linux-i586.rpm
Préparation...              ########################################### [100%]
   1:j2sdk                  ########################################### [100%]
[root@localhost eclipse3]#
```

Le JDK a été installé dans le répertoire /usr/java/j2sdk1.4.2_06

Pour permettre l'utilisation par tous les utilisateurs du système, le plus simple est de créer un fichier de configuration dans le répertoire /etc/profile.d

Créez un fichier java.sh

Exemple : le contenu du fichier java.sh

```
[root@localhost root]# cat java.sh
export JAVA_HOME="/usr/java/j2sdk1.4.2_06"
export PATH=$PATH:$JAVA_HOME/bin
```

Modifiez ces droits pour permettre son exécution

Exemple :

```
[root@localhost root]# chmod 777 java.sh
[root@localhost root]# source java.sh
```

Si kaffe est déjà installé sur le système il est préférable de mettre le chemin vers le JDK en tête de la variable PATH

Exemple :

```
[root@localhost root]# java
usage: kaffe [-options] class
Options are:
        -help                   Print this message
        -version                Print version number
        -fullversion            Print verbose version info
```

```
        -ss <size>              Maximum native stack size
[root@localhost root]# cat java.sh
export JAVA_HOME="/usr/java/j2sdk1.4.2_06"
export PATH=$JAVA_HOME/bin:$PATH
```

Pour rendre cette modification permanente, il faut copier le fichier java.sh dans le répertoire /etc/profile.d

Exemple :

```
[root@localhost root]# cp java.sh /etc/profile.d
```

Ainsi tous utilisateurs qui ouvriront une nouvelle console Bash aura ces variables d'environnements positionnées pour utiliser les outils du JDK.

Exemple :

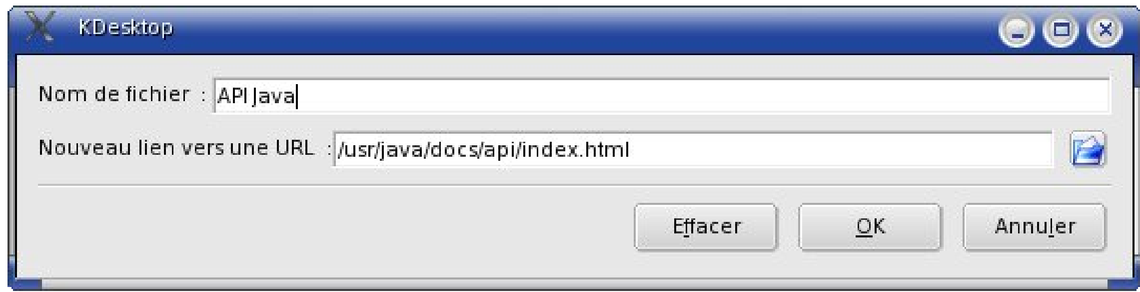
```
[java@localhost java]$ echo $JAVA_HOME
/usr/java/j2sdk1.4.2_06
[java@localhost java]$ java -version
java version "1.4.2_06"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.4.2_06-b03)
Java HotSpot(TM) Client VM (build 1.4.2_06-b03, mixed mode)
[java@localhost java]$
```

L'installation de la documentation se fait en décompressant l'archive dans un répertoire du système par exemple /usr/java.

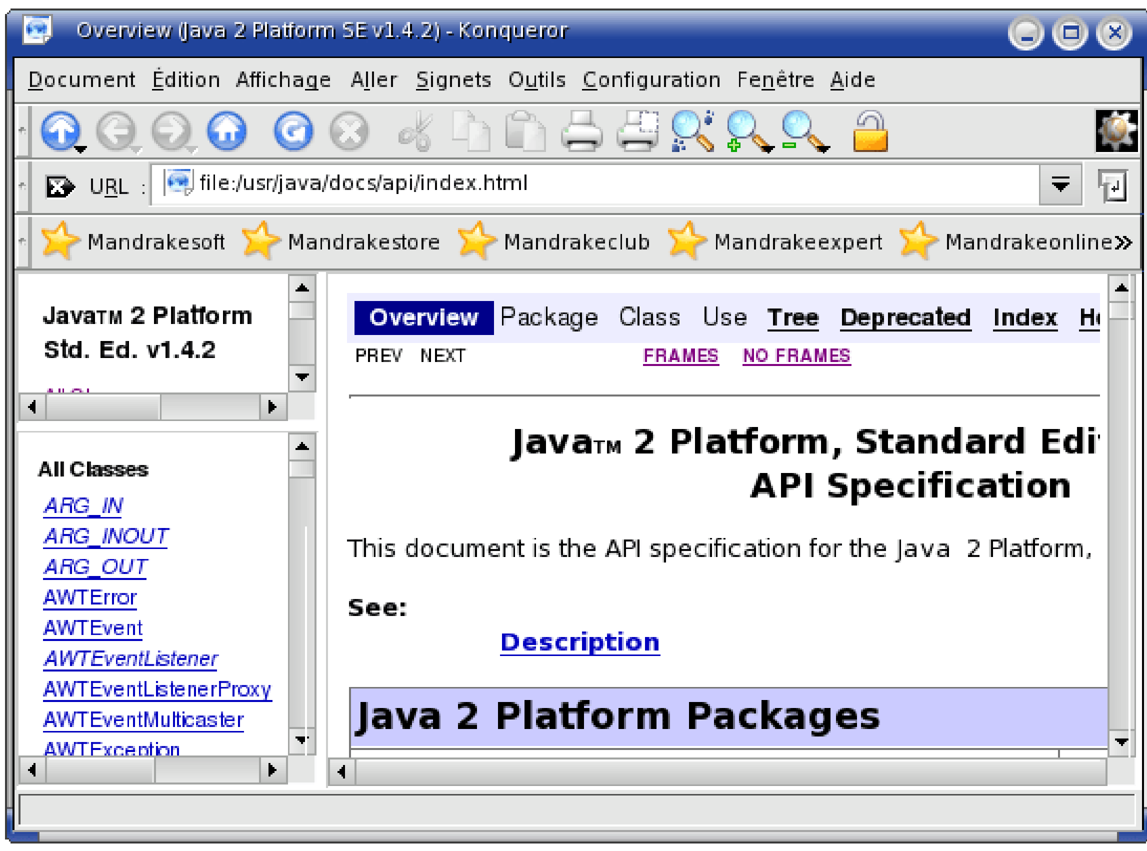
Exemple :

```
[root@localhost local]# mv j2sdk-1_4_2-doc.zip /usr/java
[root@localhost java]# ll
total 33636
drwxr-xr-x  8 root root     4096 oct 16 22:18 j2sdk1.4.2_06/
-rwxr--r--  1 root root 34397778 oct 18 23:39 j2sdk-1_4_2-doc.zip*
[root@localhost java]# unzip -q j2sdk-1_4_2-doc.zip
[root@localhost java]# ll
total 33640
drwxrwxr-x  8 root root     4096 aoû 15  2003 docs/
drwxr-xr-x  8 root root     4096 oct 16 22:18 j2sdk1.4.2_06/
-rwxr--r--  1 root root 34397778 oct 18 23:39 j2sdk-1_4_2-doc.zip*
[root@localhost java]# rm j2sdk-1_4_2-doc.zip
rm: détruire fichier régulier `j2sdk-1_4_2-doc.zip'? o
[root@localhost java]# ll
total 8
drwxrwxr-x  8 root root 4096 aoû 15  2003 docs/
drwxr-xr-x  8 root root 4096 oct 16 22:18 j2sdk1.4.2_06/
[root@localhost java]#
```

Il est possible pour un utilisateur de créer un raccourci sur le bureau KDE en utilisant le menu contextuel créer un « nouveau/fichier/lien vers une url ... »



Un double clic sur la nouvelle icône permet d'ouvrir directement Konqueror avec l'aide en ligne de l'API.


Overview (Java 2 Platform SE v1.4.2) - Konqueror
Document Édition Affichage Aller Signets Outils Configuration Fenêtre Aide
URL : file:/usr/java/docs/api/index.html
Mandrakesoft Mandrakestore Mandrakeclub Mandrakeexpert Mandrakeonline
JavaTM 2 Platform
Std. Ed. v1.4.2
All Classes
ARG_IN
ARG_INOUT
ARG_OUT
AWTError
AWTEvent
AWTEventListener
AWTEventListenerProxy
AWTEventMulticaster
Overview Package Class Use Tree Deprecated Index
PREV NEXT FRAMES NO FRAMES
API Specification
This document is the API specification for the Java 2 Platform,
See:
Description
Java 2 Platform Packages

2. Les notions et techniques de base en Java

Ce chapitre présente quelques concepts de base utilisés en Java relatifs à la compilation et l'exécution d'applications notamment la notion de classpath, de packages et d'archives de déploiement jar.

Ce chapitre contient plusieurs sections :

- Les concepts de base
- L'exécution d'une applet : cette section présente l'exécution d'un programme et d'une applet.

2.1. Les concepts de base

La plate-forme Java utilise quelques notions base dans sa mise en oeuvre notamment :

- La compilation du code source dans un langage indépendant de la plate-forme d'exécution : le byte code
- l'exécution du byte code par une machine virtuelle nommée JVM (Java Virtual Machine)
- la notion de package qui permet d'organiser les classes
- le classpath qui permet de préciser au compilateur et à la JVM où elle peut trouver les classes requises par l'application
- le packaging des classes compilés dans une archive de déploiement nommé jar (Java ARchive)

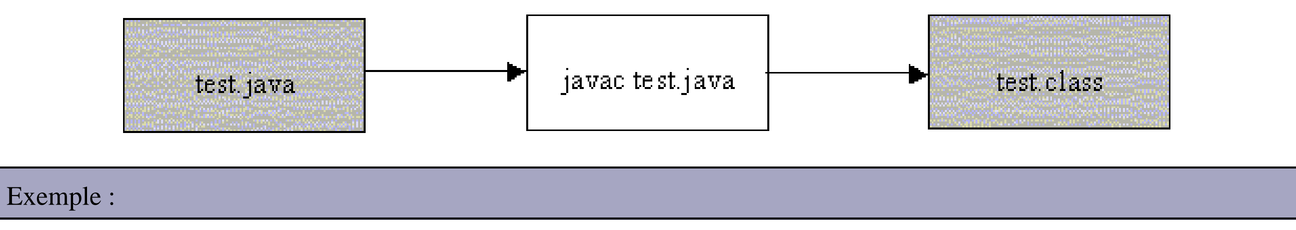
2.1.1. La compilation et l'exécution

Un programme Java est composé d'un ou plus généralement plusieurs fichiers source. N'importe quel éditeur de texte peut être utilisé pour éditer un fichier source Java.

Ces fichiers source possèdent l'extension .java. Ils peuvent contenir une ou plusieurs classes ou interfaces mais il ne peut y avoir qu'une seule classe ou interface déclarée publique par fichier. Le nom de ce fichier source doit obligatoirement correspondre à la casse prêt au nom de cette entité publique suivi de l'extension .java

Il est nécessaire de compiler le source pour le transformer en J-code ou byte-code Java qui sera lui exécuté par la machine virtuelle. Pour être compilé, le programme doit être enregistré au format de caractères Unicode : une conversion automatique est faite par le JDK si nécessaire.

Un compilateur Java, par exemple l'outil javac fourni avec le JDK est utilisé pour compiler chaque fichier source en fichier de classe possédant l'extension .class. Cette compilation gère pour chaque fichier source un ou plusieurs fichiers .class qui contiennent du byte code.



Exemple :

```
public class MaClasse {

  public static void main(String[] args) {
    System.out.println("Bonjour");
  }

}
```

Résultat :

```
C:\TEMP>javac MaClasse.java

C:\TEMP>dir MaClas*
 Volume in drive C has no label.
 Volume Serial Number is 1E06-2R43

 Directory of C:\TEMP

31/07/2007  13:34               417 MaClasse.class
31/07/2007  13:34               117 MaClasse.java
```

Le compilateur génère autant de fichier .class que de classes et interfaces définies dans chaque fichier source.

Exemple :

```
public class MaClasse {

  public static void main(String[] args) {
    System.out.println("Bonjour");
  }

}

class MonAutreClasse {

  public static void afficher(String message) {
    System.out.println(message);
  }

}
```

Résultat :

```
C:\TEMP>dir *.class
 Volume in drive C has no label.
 Volume Serial Number is 1E06-2R43

 Directory of C:\TEMP

31/07/2007  13:40               417 MaClasse.class
31/07/2007  13:40               388 MonAutreClasse.class
```

Pour exécuter une application, la classe servant de point d'entrée doit obligatoirement contenir une méthode ayant la signature public static void main(String[] args). Il est alors possible de fournir cette classe à la JVM qui va charger le ou les fichiers .class utiles à l'application et exécuter le code.

Exemple :

```
C:\TEMP>java MaClasse
Bonjour
```

Pour les classes anonymes, le compilateur génère un nom de fichier constitué du nom de la classe englobante suffixée par $ et un numéro séquentiel.

Exemple :

```
import javax.swing.JFrame;
import java.awt.event.*;

public class MonApplication {

  public static void main(String[] args) {
    MaFenetre f = new MaFenetre();
    f.afficher();
  }
}

class MaFenetre {

  JFrame mainFrame = null;

  public MaFenetre() {

    mainFrame = new JFrame();
    mainFrame.setTitle("Mon application");

    mainFrame.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent ev) {
        System.exit(0);
      }
    });

    mainFrame.setSize(320, 240);
  }

  public void afficher() {
    mainFrame.setVisible(true);
  }
}
```

Résultat :

```
C:\TEMP>javac MonApplication.java

C:\TEMP>dir *.class
 Volume in drive C has no label.
 Volume Serial Number is 1E06-2R43

 Directory of C:\TEMP

31/07/2007  13:50               494 MaFenetre$1.class
31/07/2007  13:50               687 MaFenetre.class
31/07/2007  13:50               334 MonApplication.class
```

Une classe anonyme peut elle-même définir une classe : dans ce cas le nom du fichier de classe sera celui de la classe anonyme suffixé par le caractère $ et le nom de la classe

Exemple :

```
import javax.swing.JFrame;
import java.awt.event.*;

public class MonApplication {

  public static void main(String[] args) {
    MaFenetre f = new MaFenetre();
    f.afficher();
  }
}

class MaFenetre {

  JFrame mainFrame = null;
```

```
  public MaFenetre() {

    mainFrame = new JFrame();
    mainFrame.setTitle("Mon application");

    mainFrame.addWindowListener(new WindowAdapter() {

      class MonAutreClasse {

        public void afficher(String message) {
          System.out.println(message);
        }

      }

      public void windowClosing(WindowEvent ev) {
        System.exit(0);
      }
    });

    mainFrame.setSize(320, 240);
  }

  public void afficher() {
    mainFrame.setVisible(true);
  }
}
```

Résultat :

```
C:\TEMP>javac MonApplication.java

C:\TEMP>dir *.class
 Volume in drive C has no label.
 Volume Serial Number is 1E06-2R43

 Directory of C:\TEMP

31/07/2007  13:53               549 MaFenetre$1$MonAutreClasse.class
31/07/2007  13:53               555 MaFenetre$1.class
31/07/2007  13:53               687 MaFenetre.class
31/07/2007  13:53               334 MonApplication.class
```

2.1.2. Les packages

Les fichiers sources peuvent être organisés en package. Les packages définissent une hiérarchie de noms, chaque nom étant séparé par le caractère point. Le nom d'un package est lié à une arborescence de sous répertoire correspondant à ce nom.

Ceci permet de structurer les sources d'une application car une application peut rapidement contenir plusieurs centaines voir milliers de fichiers source. Les packages permettent aussi d'assurer l'unicité d'une classe grâce à son nom pleinement qualifié (nom du package suivi du caractère «.» suivi du nom de la classe).

L'API Java est organisée en packages répartis en trois grands ensembles :

- Packages standards : ce sont les sous packages du package java
- Packages d'extensions : ce sont les sous packages du package javax
- Packages tiers : ces packages concernant notamment Corba et XML

Les principaux packages standards de Java 6 sont :

java.applet	Création d'applets
java.awt	Création d'interfaces graphiques avec AWT

java.io	Accès aux flux entrant et sortant
java.lang	Classes et interfaces fondamentales
java.math	Opérations mathématiques
java.net	Accès aux réseaux
java.nio	API NIO
java.rmi	API RMI (invocation de méthodes distantes)
java.security	Mise en oeuvre de fonctionnalités concernant la sécurité
java.sql	API JDBC (accès aux bases de données)
java.util	Utilitaires (collections, internationalisation, logging, expressions régulières,...).

Les principaux packages d'extensions de Java 6 sont :

javax.crypto	Cryptographie
javax.jws	Services web
javax.management	API JMX
javax.naming	API JNDI (Accès aux annuaires)
javax.rmi	RMI-IIOP
javax.script	API Scripting
javax.security	Authentification et habilitations
javax.swing	API Swing pour le développement d'interfaces graphiques
javax.tools	Api pour l'accès à certains outils comme le compilateur par exemple
javax.xml.bind	API JAXB pour la mapping objet/XML
javax.xml.soap	Création de messages SOAP
javax.xml.stream	API StAX (traitement de documents XML)
javax.xml.transform	Transformation de documents XML
javax.xml.validation	Validation de document XML
javax.xml.ws	API JAX-WS (service web)

Les principaux packages tiers de Java 6 sont :

org.omg.CORBA	Mise en oeuvre de CORBA
org.w3c.dom	Traitement de documents XML avec DOM
org.xml.sax	Traitement de documents XML avec SAX

Le package est précisé dans le fichier source grâce à l'instruction package. Le fichier doit donc dans ce cas être stocké dans une arborescence de répertoires qui correspond au nom du package.

Exemple :

```
package com.jmdoudoux.test;

public class MaCLasseTest {
```

```
  public static void main() {
      System.out.println("Bonjour");
  }

}
```

Si les sources de l'application sont dans le répertoire C:\Documents and Settings\jm\workspace\Tests, alors le fichier MaCLasseTest.java doit être dans le répertoire C:\Documents and Settings\jm\workspace\Tests\com\jmdoudoux\test.

Si aucun package n'est précisé, alors c'est le package par défaut (correspondant au répertoire courant) qui est utilisé. Ce n'est pas une bonne pratique d'utiliser le package par défaut sauf pour des tests.

Dans le code source, pour éviter d'avoir à utiliser les noms pleinement qualifiés des classes, il est possible d'utiliser l'instruction import suivi d'un nom de package suivi d'un caractère «.» et du nom d'une classe ou du caractère «*»

Exemple :

```
import javax.swing.JFrame;
import java.awt.event.*;
```

Remarque : par défaut le package java.lang est toujours importé par le compilateur.

2.1.3. Le déploiement sous la forme d'un jar

Il est possible de créer une enveloppe qui va contenir tous les fichiers d'une application Java ou une portion de cette application dans un fichier .jar (Java archive). Ceci inclus : l'arborescence des packages, les fichiers .class, les fichiers de ressources (images, configuration, ...), ... Un fichier .jar est physiquement une archive de type Zip qui contient tous ces éléments.

L'outil jar fourni avec le jdk permet de manipuler les fichiers jar.

Exemple :

```
C:\TEMP>jar cvf MonApplication.jar *.class
manifest ajoutÚ
ajout : MaFenetre$1$MonAutreClasse.class (entrÚe = 549) (sortie = 361) (34% comp
ressÚs)
ajout : MaFenetre$1.class (entrÚe = 555) (sortie = 368) (33% compressÚs)
ajout : MaFenetre.class (entrÚe = 687) (sortie = 467) (32% compressÚs)
ajout : MonApplication.class (entrÚe = 334) (sortie = 251) (24% compressÚs)
```

Le fichier .jar peut alors être diffusé et exécuté si il contient au moins une classe avec une méthode main().

Exemple : déplacement du jar pour être sûre qu'il n'utilise pas de classe du répertoire et exécution

```
C:\TEMP>copy MonApplication.jar ..
        1 file(s) copied.

C:\TEMP>cd ..

C:\>java -cp MonApplication.jar MonApplication
```

Remarque : un fichier .jar peut contenir plusieurs packages.

Le fichier jar peut inclure un fichier manifest qui permet de préciser des informations d'exécution sur le fichier jar (classe principale à exécuter, classpath, ...) : ceci permet d'exécuter directement l'application en double cliquant sur le fichier .jar.

2.1.4. Le classpath

A l'exécution, la JVM et les outils du JDK recherchent les classes requises dans :

- Les classes de la plate-forme Java (stockées dans le fichier rt.jar)
- Les classes d'extension de la plate-forme Java
- Le classpath

Important : il n'est pas recommandé d'ajouter des classes ou des bibliothèques dans les sous répertoires du JDK.

La notion de classpath est importante car elle est toujours utilisée quelque soit l'utilisation qui est fait de Java (ligne de commandes, IDE, script Ant, ...). Le classpath est sûrement la notion de base en Java qui pose le plus de soucis aux développeurs inexpérimentés en Java mais sa compréhension est absolument nécessaire.

Le classpath permet de préciser au compilateur et à la JVM où ils peuvent trouver les classes dont ils ont besoin pour la compilation et l'exécution d'une application. C'est un ensemble de chemins vers des répertoires ou des fichiers .jar dans lequel l'environnement d'exécution Java recherche les classes (celles de l'application mais aussi celles de tiers) et éventuellement des fichiers de ressources utiles à l'exécution de l'application. Ces classes ne concernent pas celles fournies par l'environnement d'exécution incluses dans le fichier rt.jar qui est implicitement utilisé par l'environnement.

Le classpath est constitué de chemins vers des répertoires et/ou des archives sous la forme de fichiers .jar ou .zip. Chaque élément du classpath peut donc être :

- Pour des fichiers .class : le répertoire qui contient l'arborescence des sous répertoires des packages ou les fichiers .class (si ceux-ci sont dans le package par défaut)
- Pour des fichiers .jar ou .zip : le chemin vers chacun des fichiers

Les éléments du classpath qui ne sont pas des répertoires ou des fichiers .jar ou .zip sont ignorés.

Ces chemins peuvent être absolus ou relatifs. Chaque chemin est séparé par un caractère spécifique au système d'exploitation utilisé : point virgule sous Windows et deux points sous Unix par exemple.

Exemple sous Windows :
`.;C:\java\tests\bin;C:\java\lib\log4j-1.2.11.jar;"C:\Program Files\tests\tests.jar"`

Dans cet exemple, le classpath est composé de quatre entités :

- le répertoire courant
- le répertoire C:\java\tests\bin
- le fichier C:\java\lib\log4j-1.2.11.jar
- le fichier C:\Program Files\tests\tests.jar qui est entouré par des caractères " parce qu'il y a un espace dans son chemin

Remarque : sous Windows, il est possible d'utiliser le caractère / ou \ comme séparateur d'arborescence de répertoires.

Par défaut, si aucun classpath n'est défini, le classpath est composé uniquement du répertoire courant. Une redéfinition du classpath (avec l'option -classpath ou -cp ou la variable d'environnement système CLASSPATH) inhibe cette valeur par défaut.

La recherche d'une classe se fait dans l'ordre des différents chemins du classpath : cet ordre est donc important surtout si une bibliothèque est précisée dans deux chemins. Dans ce cas, c'est le premier trouvé dans l'ordre précisé qui sera utilisé, ce qui peut être à l'origine de problèmes.

Le classpath peut être défini à plusieurs niveaux :

1. Au niveau global : il faut utiliser la variable d'environnement système CLASSPATH

Exemple sous Windows

Il faut utiliser la commande set pour définir la variable d'environnement CLASSPATH. Le séparateur entre chaque élément du classpath est le caractère point virgule. Il ne faut pas mettre d'espace entre le signe égal.

Exemple :

```
set CLASSPATH=C:\java\classes;C:\java\lib;C:\java\lib\mysql.jar;.
```

Sous Windows 9x : il est possible ajouter une ligne définissant la variable d'environnement dans le fichier autoexec.bat :

Exemple :

```
set CLASSPATH=.;c:\java\lib\mysql.jar;%CLASSPATH%
```

Sous Windows NT/2000/XP : il faut lancer l'application démarrer/paramètre/panneau de configuration/système, ouvrir l'onglet "avancé" et cliquer sur le bouton "Variables d'environnement". Il faut ajouter ou modifier la variable CLASSPATH avec comme valeur les différents éléments du classpath séparés chacun par un caractère point virgule

Exemple sous Unix (interpréteur bash)

Exemple :

```
CLASSPATH=.:./lib/log4j-1.2.11.jar
export CLASSPATH;
```

2. Au niveau spécifique : en utilisant l'option -classpath ou -cp du compilateur javac et de la machine virtuelle
3. Au niveau d'un script de lancement : cela permet de définir la variable d'environnement CLASSPATH uniquement pour le contexte d'exécution du script

L'utilisation de la variable système CLASSPATH est pratique car elle évite d'avoir à définir le classpath pour compiler ou exécuter mais c'est une mauvaise pratique car cela peut engendrer des problèmes :

- peut vite devenir un casse tête lorsque le nombre d'applications augmente
- ce n'est pas portable d'une machine à une autre
- peut engendrer des conflits de version entre applications où entre bibliothèques

Si la JVM ou le compilateur n'arrive pas à trouver une classe dans le classpath, une exception de type java.lang.ClassNotFoundException à la compilation ou java.lang.NoClassDefFoundError à l'exécution est levée.

Exemple :

```
package com.jmdoudoux.test;

public class MaCLasseTest {

  public static void main() {
      System.out.println("Bonjour");
  }

}
```

Le fichier MaCLassTest.class issu de la compilation est stocké dans le répertoire C:\Documents and Settings\jm\workspace\Tests\com\jmdoudoux\test

En débutant en Java, il est fréquent de se placer dans le répertoire qui contient le fichier .class et de lancer la JVM.

Exemple :

```
C:\Documents and Settings\jmd\workspace\Tests\com\jmdoudoux\test>java MaCLasseTe
st
Exception in thread "main" java.lang.NoClassDefFoundError: MaCLasseTest (wrong n
```

```
ame: com/jmdoudoux/test/MaCLasseTest)
        at java.lang.ClassLoader.defineClass1(Native Method)
        at java.lang.ClassLoader.defineClass(Unknown Source)
        at java.security.SecureClassLoader.defineClass(Unknown Source)
        at java.net.URLClassLoader.defineClass(Unknown Source)
        at java.net.URLClassLoader.access$000(Unknown Source)
        at java.net.URLClassLoader$1.run(Unknown Source)
        at java.security.AccessController.doPrivileged(Native Method)
        at java.net.URLClassLoader.findClass(Unknown Source)
        at java.lang.ClassLoader.loadClass(Unknown Source)
        at sun.misc.Launcher$AppClassLoader.loadClass(Unknown Source)
        at java.lang.ClassLoader.loadClass(Unknown Source)
        at java.lang.ClassLoader.loadClassInternal(Unknown Source)
```

Cela ne fonctionne pas car la JVM cherche à partir du répertoire courant (défini dans le classpath par défaut) une classe qui soit définie dans le package par défaut (aucun nom de package précisé). Hors dans l'exemple, la classe est défini dans le package com.jmdoudoux.test.

Une autre erreur assez fréquente est de se déplacer dans le répertoire qui contient le premier répertoire du package

Exemple :

```
C:\Documents and Settings\jm\workspace\Tests\com\jmdoudoux\test>cd ../../..

C:\Documents and Settings\jm\workspace\Tests>java MaCLasseTest
Exception in thread "main" java.lang.NoClassDefFoundError: MaCLasseTest
```

Dans ce cas, cela ne fonctionne pas car le nom de la classe n'est pas pleinement qualifié

Exemple :

```
C:\Documents and Settings\jmd\workspace\Tests>java com.jmdoudoux.test.MaCL
asseTest
Bonjour
```

En précisant le nom pleinement qualifié de la classe, l'application est exécutée.

Si le classpath est redéfini, il ne faut pas oublier d'ajouter le répertoire courant au besoin en utilisant le caractère point. Cette pratique n'est cependant pas recommandée.

Exemple :

```
C:\Documents and Settings\jmd\workspace\Tests>java -cp test.jar com.jmdoudoux.te
st.MaCLasseTest
Exception in thread "main" java.lang.NoClassDefFoundError: com/jmdoudoux/test/Ma
CLasseTest

C:\Documents and Settings\jmd\workspace\Tests>java -cp test.jar;. com.jmdoudoux.
test.MaCLasseTest
Bonjour
```

Les IDE fournissent tous des facilités pour gérer le classpath. Cependant en débutant, il est préférable d'utiliser les outils en ligne de commande pour bien comprendre le fonctionnement du classpath.

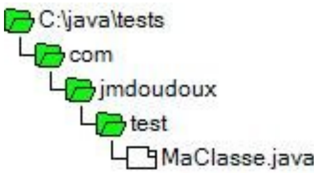
2.1.4.1. La définition du classpath pour exécuter une application

Dans cette section, une application est contenue dans le répertoire c:\java\tests. Elle est composée de la classe com.jmdoudoux.test.MaClasse.java.

Exemple :

```
package com.jmdoudoux.test;

public class MaClasse {

  public static void main(
      String[] args) {
    System.out.println("Bonjour");
  }

}
```

La structure des répertoires et fichiers de l'application est la suivante :

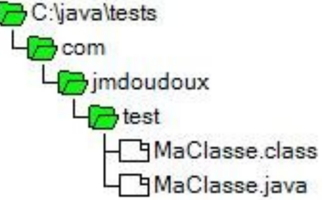


Pour compiler la classe MaClasse, il faut utiliser la commande :

Exemple :

```
C:\java\tests>javac com/jmdoudoux/test/MaClasse.java
```

Le fichier MaClasse.class est créé



Pour exécuter la classe, il faut utiliser la commande

Exemple :

```
C:\java\tests>java com.jmdoudoux.test.MaClasse
Bonjour
```

Remarque : il est inutile de spécifier le classpath puisque celui ci n'est composé que du répertoire courant qui correspond au classpath par défaut.

Il est cependant possible de le préciser explicitement

Exemple :

```
C:\java\tests>java -cp . com.jmdoudoux.test.MaClasse
Bonjour

C:\java\tests>java -cp c:/java/tests com.jmdoudoux.test.MaClasse
Bonjour
```

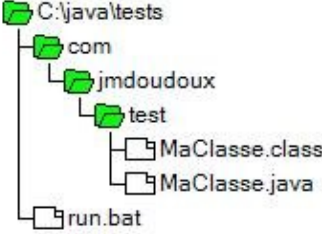
2.1.4.2. La définition du classpath pour exécuter une application avec la variable CLASSPATH

Il est possible de définir le classpath en utilisant la variable d'environnement système CLASSPATH.

Exemple : le fichier run.bat

```
@echo off
set CLASSPATH=c:/java/tests
javac com/jmdoudoux/test/MaClasse.java
java com.jmdoudoux.test.MaClasse
```

Ce script redéfinit la variable CLASSPATH, exécute le compilateur javac et l'interpréteur java pour exécuter la classe. Ces deux commandes utilisent la variable CLASSPATH.



Exemple :

```
C:\java\tests>run.bat
Bonjour
```

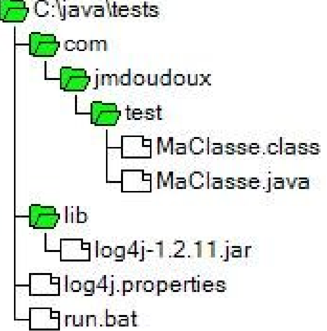
2.1.4.3. La définition du classpath pour exécuter une application utilisant une ou plusieurs bibliothèques

L'exemple de cette section va utiliser la bibliothèque log4j.

Exemple :

```
package com.jmdoudoux.test;

import org.apache.log4j.*;

public class MaClasse {
  static Logger logger = Logger.getLogger(MaClasse.class);

  public static void main(
      String[] args) {
    PropertyConfigurator.configure("log4j.properties");

    logger.info("Bonjour");
  }

}
```

Le fichier jar de log4j est stocké dans le sous répertoire lib. Le fichier de configuration log4.properties est dans le répertoire principal de l'application puisqu'il est inclus dans le classpath



Il est nécessaire de préciser dans le classpath le répertoire tests et le fichier jar de log4j.

Exemple :

```
C:\java\tests>javac -cp c:/java/tests;c:/java/tests/lib/log4j-1.2.11.jar com/jmd
oudoux/test/MaClasse.java
C:\java\tests>java -cp c:/java/tests;c:/java/tests/lib/log4j-1.2.11.jar com.jmdo
udoux.test.MaClasse
[main] INFO com.jmdoudoux.test.MaClasse - Bonjour
```

Il est aussi possible d'utiliser la variable d'environnement système classpath.

2.1.4.4. La définition du classpath pour exécuter une application packagée en jar

Il est possible de préciser les bibliothèques requises dans le fichier manifest du fichier jar.

La propriété JAR-class-path va étendre le classpath mais uniquement pour les classes chargé à partir du jar. Les classes incluses dans le JAR-class-path sont chargées comme si elles étaient incluses dans le jar.

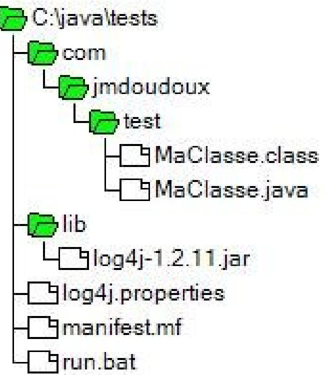
Exemple : le fichier manifest.mf

```
Main-Class: com.jmdoudoux.test.MaClasse
Class-Path: lib/log4j-1.2.11.jar
```

La clé Class-Path permet de définir le classpath utilisé lors de l'exécution.

Remarques importantes : Il faut obligatoirement que le fichier manifest ce termine par une ligne vide. Pour préciser plusieurs entités dans le classpath, il faut les séparer par un caractère espace.

La structure des répertoires et des fichiers est la suivante :

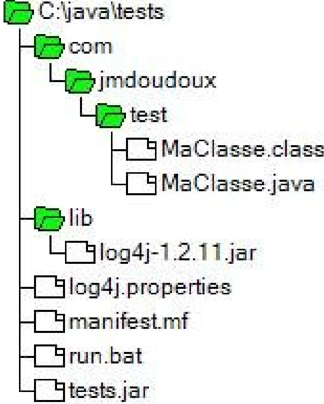


Pour créer l'archive jar, il faut utiliser l'outil jar en précisant les options de création, le nom du fichier .jar, le fichier manifest et les entités à inclure dans le fichier jar.

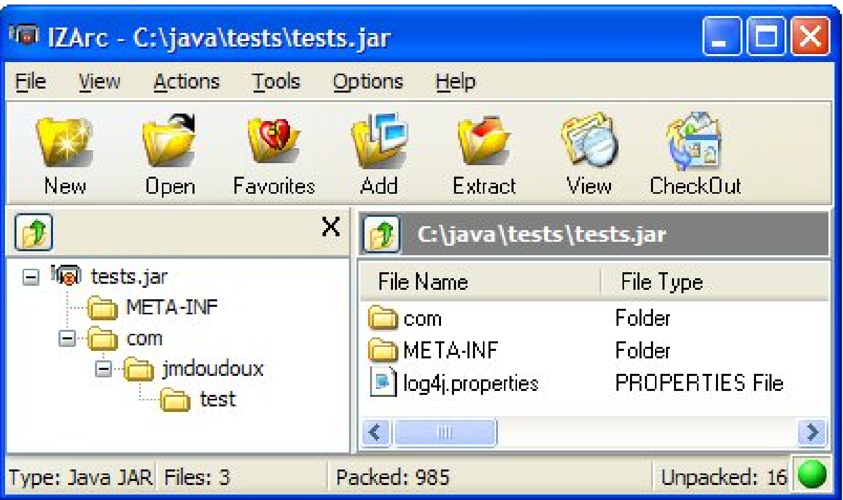
Exemple :

```
C:\java\tests>jar cfm tests.jar manifest.mf com log4j.properties
```

Le fichier jar est créé



L'archive jar ne contient pas le sous répertoire lib, donc il n'inclus pas la bibliothèque requise.



Pour exécuter l'application, il suffit d'utiliser l'interpréteur java avec l'option -jar

Exemple :

```
C:\java\tests>java -jar tests.jar
[main] INFO com.jmdoudoux.test.MaClasse - Bonjour
```

Attention : les entités précisées dans le classpath du fichier manifest doivent exister pour permettre l'exécution de l'application.

Exemple :

```
C:\java\tests>rename lib libx

C:\java\tests>java -jar tests.jar
Exception in thread "main" java.lang.NoClassDefFoundError: org/apache/log4j/Logg
er
        at com.jmdoudoux.test.MaClasse.<clinit>(MaClasse.java:6)
Caused by: java.lang.ClassNotFoundException: org.apache.log4j.Logger
        at java.net.URLClassLoader$1.run(Unknown Source)
        at java.security.AccessController.doPrivileged(Native Method)
        at java.net.URLClassLoader.findClass(Unknown Source)
        at java.lang.ClassLoader.loadClass(Unknown Source)
```

```
        at sun.misc.Launcher$AppClassLoader.loadClass(Unknown Source)
        at java.lang.ClassLoader.loadClass(Unknown Source)
        at java.lang.ClassLoader.loadClassInternal(Unknown Source)
        ... 1 more
```

2.2. L'exécution d'une applet

Il suffit de créer une page HTML pouvant être très simple :

Exemple :

```
<HTML>
<TITLE> test applet Java </TITLE>
<BODY>
<APPLET code="NomFichier.class" width="270" height="200">
</APPLET>
</BODY>
</HTML>
```

Il faut ensuite visualiser la page créée dans l'appletviewer ou dans un navigateur 32 bits compatible avec la version de Java dans laquelle l'applet est écrite.

3. La syntaxe et les éléments de bases de Java

Ce chapitre contient plusieurs sections :

- Les règles de base : cette section présente les règles syntaxiques de base de Java.
- Les identificateurs : cette section présente les règles de composition des identificateurs.
- Les commentaires : cette section présente les différentes formes de commentaires de Java.
- La déclaration et l'utilisation de variables : cette section présente la déclaration des variables, les types élémentaires, les formats des type élémentaires, l'initialisation des variables, l'affectation et les comparaisons.
- Les opérations arithmétiques : cette section présente les opérateurs arithmétique sur les entiers et les flottants et les opérateurs d'incrémentation et de décrémentation.
- La priorité des opérateurs : cette section présente la priorité des opérateurs.
- Les structures de contrôles : cette section présente les instructions permettant la réalisation de boucles, de branchements conditionnels et de débranchements.
- Les tableaux : cette section présente la déclaration, l'initialisation explicite et le parcours d'un tableau
- Les conversions de types : cette section présente la conversion de types élémentaires.
- La manipulation des chaînes de caractères : cette section présente la définition et la manipulation de chaîne de caractères (addition, comparaison, changement de la casse ...).

3.1. Les règles de base

Java est sensible à la casse.

Les blocs de code sont encadrés par des accolades. Chaque instruction se termine par un caractère ';' (point virgule).

Une instruction peut tenir sur plusieurs lignes :

Exemple :

```
char
code
=
'D';
```

L'indentation est ignorée du compilateur mais elle permet une meilleure compréhension du code par le programmeur.

3.2. Les identificateurs

Chaque objet, classe, programme ou variable est associé à un nom : l'identificateur qui peut se composer de tous les caractères alphanumériques et des caractères _ et $. Le premier caractère doit être une lettre, le caractère de soulignement ou le signe dollar.

Rappel : Java est sensible à la casse.

Un identificateur ne peut pas appartenir à la liste des mots réservé du langage Java :

abstract	const	final	int	public	throw
assert(Java 1.4)	continue	finally	interface	return	throws
boolean	default	float	long	short	transient
break	do	for	native	static	true
byte	double	goto	new	strictfp	try
case	else	if	null	super	void
catch	enum (Java 5)	implements	package	switch	volatile
char	extends	import	private	synchronized	while
class	false	instanceof	protected	this	

3.3. Les commentaires

Ils ne sont pas pris en compte par le compilateur donc ils ne sont pas inclus dans le pseudo code. Ils ne se terminent pas par un caractère ";".

Il existe trois types de commentaire en Java :

Type de commentaires	Exemple
commentaire abrégé	`// commentaire sur une seule ligne` `int N=1; // déclaration du compteur`
commentaire multi ligne	`/* commentaires ligne 1` `     commentaires ligne 2 */`

commentaire de documentation automatique	`/**`
	`  * commentaire de la methode`
	`  * @param val la valeur a traiter`
	`  * @since 1.0`
	`  * @return Rien`
	`  * @deprecated Utiliser la nouvelle methode XXX`
	`  */`

3.4. La déclaration et l'utilisation de variables

3.4.1. La déclaration de variables

Une variable possède un nom, un type et une valeur. La déclaration d'une variable doit donc contenir deux choses : un nom et le type de données qu'elle peut contenir. Une variable est utilisable dans le bloc ou elle est définie.

La déclaration d'une variable permet de réserver la mémoire pour en stocker la valeur.

Le type d'une variable peut être :

- soit un type élémentaire dit aussi type primitif déclaré sous la forme type_élémentaire variable;
- soit une classe déclarée sous la forme classe variable ;

Exemple :

```
long nombre;
int compteur;
String chaine;
```

Rappel : les noms de variables en Java peuvent commencer par une lettre, par le caractère de soulignement ou par le signe dollar. Le reste du nom peut comporter des lettres ou des nombres mais jamais d'espaces.

Il est possible de définir plusieurs variables de même type en séparant chacune d'elles par une virgule.

Exemple :

```
int jour, mois, annee ;
```

Java est un langage à typage rigoureux qui ne possède pas de transtypage automatique lorsque ce transtypage risque de conduire à une perte d'information.

Pour les objets, il est nécessaire en plus de la déclaration de la variable de créer un objet avant de pouvoir l'utiliser. Il faut réserver de la mémoire pour la création d'un objet (remarque : un tableau est un objet en Java) avec l'instruction new. La libération de la mémoire se fait automatiquement grâce au garbage collector.

Exemple :

```
MaClasse instance; // déclaration de l'objet

instance = new MaClasse(); // création de l'objet

OU MaClasse instance = new MaClasse(); // déclaration et création de l'objet
```

Exemple :

```
int[] nombre = new int[10];
```

Il est possible en une seule instruction de faire la déclaration et l'affectation d'une valeur à une variable ou plusieurs variables.

Exemple :

```
int i=3 , j=4 ;
```

3.4.2. Les types élémentaires

Les types élémentaires ont une taille identique quelque soit la plate-forme d'exécution : c'est un des éléments qui permet à Java d'être indépendant de la plate-forme sur lequel le code s'exécute.

Type	Désignation	Longueur	Valeurs	Commentaires
boolean	valeur logique : true ou false	1 bit	true ou false	pas de conversion possible vers un autre type
byte	octet signé	8 bits	-128 à 127	
short	entier court signé	16 bits	-32768 à 32767	
char	caractère Unicode	16 bits	\u0000 à \uFFFF	entouré de cotes simples dans du code Java
int	entier signé	32 bits	-2147483648 à 2147483647	

float	virgule flottante simple précision (IEEE754)	32 bits	1.401e-045 à 3.40282e+038	
double	virgule flottante double précision (IEEE754)	64 bits	2.22507e-308 à 1.79769e+308	
long	entier long	64 bits	-9223372036854775808 à 9223372036854775807	

Les types élémentaires commencent tous par une minuscule.

3.4.3. Le format des types élémentaires

Le format des nombres entiers :

Il existe plusieurs formats pour les nombres entiers : les types byte, short, int et long peuvent être codés en décimal, hexadécimal ou octal. Pour un nombre hexadécimal, il suffit de préfixer sa valeur par 0x. Pour un nombre octal, le nombre doit commencer par un zéro. Le suffixe l ou L permet de spécifier que c'est un entier long.

Le format des nombres décimaux :

Il existe plusieurs formats pour les nombres décimaux : les types float et double stockent des nombres flottants : pour être reconnus comme tel ils doivent posséder soit un point, un exposant ou l'un des suffixes f, F, d, D. Il est possible de préciser des nombres qui n'ont pas le partie entière ou pas de partie décimale.

Exemple :

```
float pi = 3.141f;
double valeur = 3d;
float flottant1 = +.1f , flottant2 = 1e10f;
```

Par défaut un littéral représentant une valeur décimale est de type double : pour définir un littéral représentant une valeur décimale de type float il faut le suffixer par la lettre f ou F.

Attention :

```
float pi = 3.141; // erreur à la compilation
float pi = 3.141f; // compilation sans erreur
```

Exemple :

```
double valeur = 1.1;
```

Le format des caractères :

Un caractère est codé sur 16 bis car il est conforme à la norme Unicode. Il doit être entouré par des apostrophes. Une valeur de type char peut être considérée comme un entier non négatif de 0 à 65535. Cependant la conversion implicite par affectation n'est pas possible.

Exemple :

```
/* test sur les caractères */
class test1 {
    public static void main (String args[]) {
        char code = 'D';
        int index = code - 'A';
        System.out.println("index = " + index);
    }
```

```
}
```

3.4.4. L'initialisation des variables

Exemple :

```
int nombre; // déclaration
nombre = 100; //initialisation
OU int nombre = 100; //déclaration et initialisation
```

En Java, toute variable appartenant à un objet (définie comme étant un attribut de l'objet) est initialisée avec une valeur par défaut en accord avec son type au moment de la création. Cette initialisation ne s'applique pas aux variables locales des méthodes de la classe.

Les valeurs par défaut lors de l'initialisation automatique des variables d'instances sont :

Type	Valeur par défaut
boolean	false
byte, short, int, long	0
float, double	0.0
char	\u000
classe	null

Remarque : Dans une applet, il est préférable de faire les déclarations et initialisation dans la méthode init().

3.4.5. L'affectation

le signe = est l'opérateur d'affectation et s'utilise avec une expression de la forme variable = expression. L'opération d'affectation est associative de droite à gauche : il renvoie la valeur affectée ce qui permet d'écrire :

```
x = y = z = 0;
```

Il existe des opérateurs qui permettent de simplifier l'écriture d'une opération d'affectation associée à un opérateur mathématique :

Opérateur	Exemple	Signification
=	a=10	équivalent à : a = 10
+=	a+=10	équivalent à : a = a + 10
-=	a-=	équivalent à : a = a - 10
*=	a*=	équivalent à : a = a * 10
/=	a/=10	équivalent à : a = a / 10
%=	a%=10	reste de la division
^=	a^=10	équivalent à : a = a ^ 10
<<=	a<<=10	équivalent à : a = a << 10 a est complété par des zéros à droite
>>=	a>>=10	équivalent à : a = a >> 10 a est complété par des zéros à gauche
>>>=	a>>>=10	équivalent à : a = a >>> 10 décalage à gauche non signé

Attention : Lors d'une opération sur des opérandes de types différents, le compilateur détermine le type du résultat en prenant le type le plus précis des opérandes. Par exemple, une multiplication d'une variable de type float avec une variable de type double donne un résultat de type double. Lors d'une opération entre un opérande entier et un flottant, le résultat est du type de l'opérande flottant.

3.4.6. Les comparaisons

Java propose des opérateurs pour toutes les comparaisons :

Opérateur	Exemple	Signification
>	a > 10	strictement supérieur
<	a < 10	strictement inférieur
>=	a >= 10	supérieur ou égal
<=	a <= 10	inférieur ou égal
==	a == 10	Egalité
!=	a != 10	diffèrent de
&	a & b	ET binaire
^	a ^ b	OU exclusif binaire
|	a | b	OU binaire
&&	a && b	ET logique (pour expressions booléennes) : l'évaluation de l'expression cesse dès qu'elle devient fausse
||	a || b	OU logique (pour expressions booléennes) : l'évaluation de l'expression cesse dès qu'elle devient vraie
? :	a ? b : c	opérateur conditionnel : renvoie la valeur b ou c selon l'évaluation de l'expression a (si a alors b sinon c) : b et c doivent retourner le même type

Les opérateurs sont exécutés dans l'ordre suivant à l'intérieure d'une expression qui est analysée de gauche à droite:

- incréments et décréments
- multiplication, division et reste de division (modulo)
- addition et soustraction
- comparaison
- le signe = d'affectation d'une valeur à une variable

L'usage des parenthèses permet de modifier cet ordre de priorité.

3.5. Les opérations arithmétiques

Les opérateurs arithmétiques se notent + (addition), - (soustraction), * (multiplication), / (division) et % (reste de la division). Ils peuvent se combiner à l'opérateur d'affectation

Exemple :

```
nombre += 10;
```

3.5.1. L'arithmétique entière

Pour les types numériques entiers, Java met en oeuvre une sorte de mécanisme de conversion implicite vers le type int appelée promotion entière. Ce mécanisme fait partie des règles mise en place pour renforcer la sécurité du code.

Exemple :

```
short x= 5 , y = 15;
x = x + y ; //erreur à la compilation

Incompatible type for =. Explicit cast needed to convert int to short.
x = x + y ; //erreur à la compilation
^
1 error
```

Les opérandes et le résultat de l'opération sont convertis en type int. Le résultat est affecté dans un type short : il y a donc risque de perte d'informations et donc erreur à la compilation est émise. Cette promotion évite un débordement de capacité sans que le programmeur soit pleinement conscient du risque : il est nécessaire, pour régler le problème, d'utiliser une conversion explicite ou cast

Exemple :

```
x = (short) ( x + y );
```

Il est nécessaire de mettre l'opération entre parenthèse pour que ce soit son résultat qui soit converti car le cast a une priorité plus forte que les opérateurs arithmétiques.

La division par zéro pour les types entiers lève l'exception ArithmeticException

Exemple :

```
/* test sur la division par zero de nombres entiers */
class test3 {
    public static void main (String args[]) {
        int valeur=10;
        double résultat = valeur / 0;
        System.out.println("index = " + résultat);
    }
}
```

3.5.2. L'arithmétique en virgule flottante

Avec des valeurs float ou double, la division par zéro ne produit pas d'exception mais le résultat est indiqué par une valeur spéciale qui peut prendre trois états :

- indéfini : Float.NaN ou Double.NaN (not a number)
- indéfini positif : Float.POSITIVE_INFINITY ou Double.POSITIVE_INFINITY, + ∞
- indéfini négatif : Float.NEGATIVE_INFINITY ou Double.NEGATIVE_INFINITY, - ∞

Conformément à la norme IEEE754, ces valeurs spéciales représentent le résultat d'une expression invalide NaN, une valeur supérieure au plafond du type pour infini positif ou négatif.

X	Y	X / Y	X % Y
valeur finie	0	+ ∞	NaN
valeur finie	+/- ∞	0	x
0	0	NaN	NaN

+/- ∞	valeur finie	+/- ∞	NaN
+/- ∞	+/- ∞	NaN	NaN

Exemple :

```
/* test sur la division par zero de nombres flottants */

class test2 {
    public static void main (String args[]) {
        float valeur=10f;
        double résultat = valeur / 0;
        System.out.println("index = " + résultat);
    }
}
```

3.5.3. L'incrémentation et la décrémentation

Les opérateurs d'incrémentation et de décrémentation sont : `n++ ++n n-- --n`

Si l'opérateur est placé avant la variable (préfixé), la modification de la valeur est immédiate sinon la modification n'a lieu qu'à l'issu de l'exécution de la ligne d'instruction (postfixé)

L'opérateur ++ renvoie la valeur avant incrémentation s'il est postfixé, après incrémentation s'il est préfixé.

Exemple :

```
System.out.println(x++); // est équivalent à
System.out.println(x); x = x + 1;

System.out.println(++x); // est équivalent à
x = x + 1; System.out.println(x);
```

Exemple :

```
/* test sur les incrementations prefixees et postfixees */

class test4 {
    public static void main (String args[]) {
        int n1=0;
        int n2=0;
        System.out.println("n1 = " + n1 + " n2 = " + n2);
        n1=n2++;
        System.out.println("n1 = " + n1 + " n2 = " + n2);
        n1=++n2;
        System.out.println("n1 = " + n1 + " n2 = " + n2);
        n1=n1++;        //attention
        System.out.println("n1 = " + n1 + " n2 = " + n2);
    }
}
```

Résultat :

```
int n1=0;
int n2=0; // n1=0 n2=0
n1=n2++; // n1=0 n2=1
```

```
n1=++n2; // n1=2 n2=2
n1=n1++; // attention : n1 ne change pas de valeur
```

3.6. La priorité des opérateurs

Java définit les priorités dans les opérateurs comme suit (du plus prioritaire au moins prioritaire)

<table>
<tr><td>les parenthèses</td><td>( )</td></tr>
<tr><td>les opérateurs d'incrémentation</td><td>++
--</td></tr>
<tr><td>les opérateurs de multiplication, division, et modulo</td><td>*
/
%</td></tr>
<tr><td>les opérateurs d'addition et soustraction</td><td>+
-</td></tr>
<tr><td>les opérateurs de décalage</td><td><<
>></td></tr>
<tr><td>les opérateurs de comparaison</td><td><
>
<=
>=</td></tr>
<tr><td>les opérateurs d'égalité</td><td>==
!=</td></tr>
<tr><td>l'opérateur OU exclusif</td><td>^</td></tr>
<tr><td>l'opérateur ET</td><td>&</td></tr>
<tr><td>l'opérateur OU</td><td>|</td></tr>
<tr><td>l'opérateur ET logique</td><td>&&</td></tr>
<tr><td>l'opérateur OU logique</td><td>||</td></tr>
<tr><td>les opérateurs d'assignement</td><td>=
+=
-=</td></tr>
</table>

Les parenthèses ayant une forte priorité, l'ordre d'interprétation des opérateurs peut être modifié par des parenthèses.

3.7. Les structures de contrôles

Comme quasi totalité des langages de développement orienté objets, Java propose un ensemble d'instructions qui permettent de d'organiser et de structurer les traitements. L'usage de ces instructions est similaire à celui rencontré dans leur équivalent dans d'autres langages.

3.7.1. Les boucles

```
while ( boolean )
{
```

```
    ... // code a exécuter dans la boucle
}
```

Le code est exécuté tant que le booléen est vrai. Si avant l'instruction while, le booléen est faux, alors le code de la boucle ne sera jamais exécuté

Ne pas mettre de ; après la condition sinon le corps de la boucle ne sera jamais exécuté

```
do {
   ...
} while ( boolean )
```

Cette boucle est au moins exécuté une fois quelque soit la valeur du booléen;

```
for ( initialisation; condition; modification) {
    ...
}
```

Exemple :

```
for (i = 0 ; i < 10; i++ ) { ....}
for (int i = 0 ; i < 10; i++ ) { ....}
for ( ; ; ) { ... } // boucle infinie
```

L'initialisation, la condition et la modification de l'index sont optionnelles.

Dans l'initialisation, on peut déclarer une variable qui servira d'index et qui sera dans ce cas locale à la boucle.

Il est possible d'inclure plusieurs traitements dans l'initialisation et la modification de la boucle : chacun des traitements doit être séparé par une virgule.

Exemple :

```
for (i = 0 , j = 0 ; i * j < 1000;i++ , j+= 2) { ....}
```

La condition peut ne pas porter sur l'index de la boucle :

Exemple :

```
boolean trouve = false;
for (int i = 0 ; !trouve ; i++ ) {
    if ( tableau[i] == 1 )
    trouve = true;
    ... //gestion de la fin du parcours du tableau
}
```

Il est possible de nommer une boucle pour permettre de l'interrompre même si cela est peu recommandé :

Exemple :

```
int compteur = 0;
boucle:
while (compteur < 100) {

    for(int compte = 0 ; compte < 10 ; compte ++) {
        compteur += compte;
        System.out.println("compteur = "+compteur);
        if (compteur> 40) break boucle;
    }
}
```

3.7.2. Les branchements conditionnels

```
if (boolean) {
    ...
} else if (boolean) {
    ...
} else {
    ...
}
```

```
switch (expression) {
    case constante1 :
        instr11;
        instr12;
        break;

    case constante2 :
        ...
    default :
        ...
}
```

On ne peut utiliser switch qu'avec des types primitifs d'une taille maximum de 32 bits (byte, short, int, char).

Si une instruction case ne contient pas de break alors les traitements associés au case suivant sont exécutés.

Il est possible d'imbriquer des switch

L'opérateur ternaire : (condition) ? valeur-vrai : valeur-faux

Exemple :

```
if (niveau == 5) // equivalent à total = (niveau ==5) ? 10 : 5;
total = 10;
else total = 5 ;
System.out.println((sexe == " H ") ? " Mr " : " Mme ");
```

3.7.3. Les débranchements

break : permet de quitter immédiatement une boucle ou un branchement. Utilisable dans tous les contrôles de flot

continue : s'utilise dans une boucle pour passer directement à l'itération suivante

break et continue peuvent s'exécuter avec des blocs nommés. Il est possible de préciser une étiquette pour indiquer le point de retour lors de la fin du traitement déclenché par le break.

Une étiquette est un nom suivi d'un deux points qui définit le début d'une instruction.

3.8. Les tableaux

Ils sont dérivés de la classe Object : il faut utiliser des méthodes pour y accéder dont font partie des messages de la classe Object tel que equals() ou getClass().

Le premier élément d'un tableau possède l'indice 0.

3.8.1. La déclaration des tableaux

Java permet de placer les crochets après ou avant le nom du tableau dans la déclaration.

Exemple :

```
int tableau[] = new int[50]; // déclaration et allocation

OU int[] tableau = new int[50];

OU int tab[]; // déclaration
tab = new int[50]; //allocation
```

Java ne supporte pas directement les tableaux à plusieurs dimensions : il faut déclarer un tableau de tableau.

Exemple :

```
float tableau[][] = new float[10][10];
```

La taille des tableaux de la seconde dimension peut ne pas être identique pour chaque occurrence.

Exemple :

```
int dim1[][] = new int[3][];
dim1[0] = new int[4];
dim1[1] = new int[9];
dim1[2] = new int[2];
```

Chaque élément du tableau est initialisé selon son type par l'instruction new : 0 pour les numériques, '\0' pour les caractères, false pour les booléens et null pour les chaines de caractères et les autres objets.

3.8.2. L'initialisation explicite d'un tableau

Exemple :

```
int tableau[5] = {10,20,30,40,50};
int tableau[3][2] = {{5,1},{6,2},{7,3}};
```

La taille du tableau n'est pas obligatoire si le tableau est initialisé à sa création.

Exemple :

```
int tableau[] = {10,20,30,40,50};
```

Le nombre d'élément de chaque lignes peut ne pas être identique :

Exemple :

```
int[][] tabEntiers = {{1,2,3,4,5,6},
                      {1,2,3,4},
                      {1,2,3,4,5,6,7,8,9}};
```

3.8.3. Le parcours d'un tableau

Exemple :

```
for (int i = 0; i < tableau.length ; i ++) { ... }
```

La variable length retourne le nombre d'éléments du tableau.

Pour passer un tableau à une méthode, il suffit de déclarer les paramètres dans l'en tête de la méthode

Exemple :

```
public void printArray(String texte[]){ ...
}
```

Les tableaux sont toujours transmis par référence puisque ce sont des objets.

Un accès a un élément d'un tableau qui dépasse sa capacité, lève une exception du type java.lang.arrayIndexOutOfBoundsException.

3.9. Les conversions de types

Lors de la déclaration, il est possible d'utiliser un cast :

Exemple :

```
int entier = 5;
float flottant = (float) entier;
```

La conversion peut entrainer une perte d'informations.

Il n'existe pas en Java de fonction pour convertir : les conversions de type se font par des méthodes. La bibliothèque de classes API fournit une série de classes qui contiennent des méthodes de manipulation et de conversion de types élémentaires.

Classe	Rôle
String	pour les chaines de caractères Unicode
Integer	pour les valeurs entières (integer)
Long	pour les entiers longs signés (long)
Float	pour les nombres à virgules flottante (float)
Double	pour les nombres à virgule flottante en double précision (double)

Les classes portent le même nom que le type élémentaire sur lequel elles reposent avec la première lettre en majuscule.

Ces classes contiennent généralement plusieurs constructeurs. Pour y accéder, il faut les instancier puisque ce sont des objets.

Exemple :

```
String montexte;
```

```
montexte = new String("test");
```

L'objet montexte permet d'accéder aux méthodes de la classe java.lang.String

3.9.1. La conversion d'un entier int en chaîne de caractère String

Exemple :

```
int i = 10;
String montexte = new String();
montexte =montexte.valueOf(i);
```

valueOf est également définie pour des arguments de type boolean, long, float, double et char

3.9.2. La conversion d'une chaîne de caractères String en entier int

Exemple :

```
String montexte = new String(" 10 ");
Integer monnombre=new Integer(montexte);
int i = monnombre.intValue(); //conversion d'Integer en int
```

3.9.3. La conversion d'un entier int en entier long

Exemple :

```
int i=10;
Integer monnombre=new Integer(i);
long j=monnombre.longValue();
```

3.10. La manipulation des chaînes de caractères

La définition d'un caractère se fait grâce au type char :

Exemple :

```
char touche = '%';
```

La définition d'une chaîne se fait grâce à l'objet String :

Exemple :

```
String texte = " bonjour ";
```

Les variables de type String sont des objets. Partout ou des constantes chaînes de caractères figurent entre guillemets, le compilateur Java génère un objet de type String avec le contenu spécifié. Il est donc possible d'écrire :

```
String texte = " Java Java Java ".replace('a','o');
```

Les chaînes de caractères ne sont pas des tableaux : il faut utiliser les méthodes de la classe String d'un objet instancié pour effectuer des manipulations.

Il est impossible de modifier le contenu d'un objet String construit à partir d'une constante. Cependant, il est possible d'utiliser les méthodes qui renvoient une chaîne pour modifier le contenu de la chaîne.

Exemple :

```
String texte = " Java Java Java ";
texte = texte.replace('a','o');
```

Java ne fonctionne pas avec le jeu de caractères ASCII ou ANSI, mais avec Unicode (Universal Code). Ceci concerne les types char et les chaînes de caractères. Le jeu de caractères Unicode code un caractère sur plusieurs octets. Les caractères 0 à 255 correspondent exactement au jeu de caractères ASCII étendu.

3.10.1. Les caractères spéciaux dans les chaines

Dans une chaîne de caractères, plusieurs caractères particuliers doivent être utilisés avec le caractère d'échappement \. Le tableau ci dessous recense les principaux caractères.

Caractères spéciaux	Affichage
\'	Apostrophe
\"	Guillemet
\\	anti slash
\t	Tabulation
\b	retour arrière (backspace)
\r	retour chariot
\f	saut de page (form feed)
\n	saut de ligne (newline)
\0ddd	caractère ASCII ddd (octal)
\xdd	caractère ASCII dd (hexadécimal)
\udddd	caractère Unicode dddd (hexadécimal)

3.10.2. L'addition de chaînes de caractères

Java admet l'opérateur + comme opérateur de concaténation de chaines de caractères.

L'opérateur + permet de concaténer plusieurs chaînes de caractères. Il est possible d'utiliser l'opérateur +=

Exemple :

```
String texte = " ";
texte += " Hello ";
texte += " World3 ";
```

Cet opérateur sert aussi à concaténer des chaînes avec tous les types de bases. La variable ou constante est alors convertie en chaîne et ajoutée à la précédente. La condition préalable est d'avoir au moins une chaîne dans l'expression sinon le sinon '+' est évalué comme opérateur mathématique.

Exemple :

```
System.out.println(" La valeur de Pi est : "+Math.PI);
int duree = 121;
System.out.println(" durée = " +duree);
```

3.10.3. La comparaison de deux chaines

Il faut utiliser la méthode equals()

Exemple :

```
String texte1 = " texte 1 ";
String texte2 = " texte 2 ";
if ( texte1.equals(texte2) )...
```

3.10.4. La détermination de la longueur d'une chaine

La méthode length() permet de déterminer la longueur d'une chaine.

Exemple :

```
String texte = " texte ";
int longueur = texte.length();
```

3.10.5. La modification de la casse d'une chaine

Les méthodes Java toUpperCase() et toLowerCase() permettent respectivement d'obtenir une chaîne tout en majuscule ou tout en minuscule.

Exemple :

```
String texte = " texte ";
String textemaj = texte.toUpperCase();
```

4. La programmation orientée objet

L'idée de base de la programmation orientée objet est de rassembler dans une même entité appelée objet les données et les traitements qui s'y appliquent.

Ce chapitre contient plusieurs sections :

- Le concept de classe : cette section présente le concept et la syntaxe de la déclaration d'une classe
- Les objets : cette section présente la création d'un objet, sa durée de vie, le clonage d'objets, les références et la comparaison d'objets, l'objet null, les variables de classes, la variable this et l'opérateur instanceof.
- Les modificateurs d'accès : cette section présente les modificateurs d'accès des entités classes, méthodes et attributs ainsi que les mots clés qui permettent de qualifier ces entités
- Les propriétés ou attributs : cette section présente les données d'une classe : les propriétés ou attributs
- Les méthodes : cette section présente la déclaration d'une méthode, la transmission de paramètres, l'émission de messages, la surcharge, la signature d'une méthode et le polymorphisme et des méthodes particulières : les constructeurs, le destructeur et les accesseurs
- L'héritage : cette section présente l'héritage : son principe, sa mise en oeuvre, ses conséquences. Il présente aussi la redéfinition d'une méthode héritée et les interfaces
- Les packages : cette section présente la définition et l'utilisation des packages
- Les classes internes : cette section présente une extension du langage Java qui permet de définir une classe dans une autre.
- La gestion dynamique des objets : cette section présente rapidement la gestion dynamique des objets grâce à l'introspection

4.1. Le concept de classe

Une classe est le support de l'encapsulation : c'est un ensemble de données et de fonctions regroupées dans une même entité. Une classe est une description abstraite d'un objet. Les fonctions qui opèrent sur les données sont appelées des méthodes. Instancier une classe consiste à créer un objet sur son modèle. Entre classe et objet il y a, en quelque sorte, le même rapport qu'entre type et variable.

Java est un langage orienté objet : tout appartient à une classe sauf les variables de type primitives.

Pour accéder à une classe il faut en déclarer une instance de classe ou objet.

Une classe comporte sa déclaration, des variables et la définition de ses méthodes.

Une classe se compose en deux parties : un en-tête et un corps. Le corps peut être divisé en 2 sections : la déclaration des données et des constantes et la définition des méthodes. Les méthodes et les données sont pourvues d'attributs de visibilité qui gère leur accessibilité par les composants hors de la classe.

4.1.1. La syntaxe de déclaration d'une classe

La syntaxe de déclaration d'une classe est la suivante :

modificateurs class nom_de_classe [extends classe_mere] [implements interfaces] { ... }

Les modificateurs de classe (ClassModifiers) sont :

Modificateur	Rôle
abstract	la classe contient une ou des méthodes abstraites, qui n'ont pas de définition explicite. Une classe déclarée abstract ne peut pas être instanciée : il faut définir une classe qui hérite de cette classe et qui implémente les méthodes nécessaires pour ne plus être abstraite.
final	la classe ne peut pas être modifiée, sa redéfinition grâce à l'héritage est interdite. Les classes déclarées final ne peuvent donc pas avoir de classes filles.
private	la classe n'est accessible qu'à partir du fichier où elle est définie
public	La classe est accessible partout

Les modificateurs abstract et final ainsi que public et private sont mutuellement exclusifs.

Le mot clé extends permet de spécifier une superclasse éventuelle : ce mot clé permet de préciser la classe mère dans une relation d'héritage.

Le mot clé implements permet de spécifier une ou des interfaces que la classe implémente. Cela permet de récupérer quelques avantages de l'héritage multiple.

L'ordre des méthodes dans une classe n'a pas d'importance. Si dans une classe, on rencontre d'abord la méthode A puis la méthode B, B peut être appelée sans problème dans A.

4.2. Les objets

Les objets contiennent des attributs et des méthodes. Les attributs sont des variables ou des objets nécessaires au fonctionnement de l'objet. En Java, une application est un objet. La classe est la description d'un objet. Un objet est une instance d'une classe. Pour chaque instance d'une classe, le code est le même, seules les données sont différentes à chaque objet.

4.2.1. La création d'un objet : instancier une classe

Il est nécessaire de définir la déclaration d'une variable ayant le type de l'objet désiré. La déclaration est de la forme nom_de_classe nom_de_variable

Exemple :

```
MaClasse m;
String chaine;
```

L'opérateur new se charge de créer une instance de la classe et de l'associer à la variable

Exemple :

```
m = new MaClasse();
```

Il est possible de tout réunir en une seule déclaration

Exemple :

```
MaClasse m = new MaClasse();
```

Chaque instance d'une classe nécessite sa propre variable. Plusieurs variables peuvent désigner un même objet.

En Java, tous les objets sont instanciés par allocation dynamique. Dans l'exemple, la variable m contient une référence sur l'objet instancié (contient l'adresse de l'objet qu'elle désigne : attention toutefois, il n'est pas possible de manipuler ou d'effectuer des opérations directement sur cette adresse comme en C).

Si m2 désigne un objet de type MaClasse, l'instruction m2 = m ne définit pas un nouvel objet mais m et m2 désignent tous les deux le même objet.

L'opérateur new est un opérateur de haute priorité qui permet d'instancier des objets et d'appeler une méthode particulière de cet objet : le constructeur. Il fait appel à la machine virtuelle pour obtenir l'espace mémoire nécessaire à la représentation de l'objet puis appelle le constructeur pour initialiser l'objet dans l'emplacement obtenu. Il renvoie une valeur qui référence l'objet instancié.

Si l'opérateur new n'obtient pas l'allocation mémoire nécessaire, il lève l'exception OutOfMemoryError.

Remarque sur les objets de type String : un objet String est automatiquement créé lors de l'utilisation d'une constante chaîne de caractères sauf si celle ci est déjà utilisée dans la classe. Ceci permet une simplification lors de la compilation de la classe.

Exemple :

```
public class TestChaines1 {

  public static void main(String[] args) {
    String chaine1 = "bonjour";
    String chaine2 = "bonjour";
    System.out.println("(chaine1 == chaine2) = " + (chaine1 == chaine2) );
  }
}
```

Résultat :

```
(chaine1 == chaine2) = true
```

Pour obtenir une seconde instance de la chaine, il faut explicitement demander sa création en utilisant l'opérateur new.

Exemple :

```
public class TestChaines2 {

  public static void main(String[] args) {
    String chaine1 = "bonjour";
    String chaine2 = new String("bonjour");
    System.out.println("(chaine1 == chaine2) = " + (chaine1 == chaine2) );
  }
}
```

Résultat :

```
(chaine1 == chaine2) = false
```

Remarque : les tests réalisés dans ces deux exemples sont réalisés sur les références des instances. Pour tester l'égalité de la valeur des chaînes, il faut utiliser la méthode equals() de la classe String.

4.2.2. La durée de vie d'un objet

Les objets ne sont pas des éléments statiques et leur durée de vie ne correspond pas forcément à la durée d'exécution du programme.

La durée de vie d'un objet passe par trois étapes :

* la déclaration de l'objet et l'instanciation grâce à l'opérateur new

Exemple :

```
nom_de_classe nom_d_objet = new nom_de_classe( ... );
```

* l'utilisation de l'objet en appelant ces méthodes
* la suppression de l'objet : elle est automatique en Java grâce à la machine virtuelle. La restitution de la mémoire inutilisée est prise en charge par le récupérateur de mémoire (garbage collector). Il n'existe pas d'instruction delete comme en C++.

4.2.3. La création d'objets identiques

Exemple :

```
MaClasse m1 = new MaClasse();
MaClasse m2 = m1;
```

m1 et m2 contiennent la même référence et pointent donc tous les deux sur le même objet : les modifications faites à partir d'une des variables modifient l'objet.

Pour créer une copie d'un objet, il faut utiliser la méthode clone() : cette méthode permet de créer un deuxième objet indépendant mais identique à l'original. Cette méthode est héritée de la classe Object qui est la classe mère de toutes les classes en Java.

Exemple :

```
MaClasse m1 = new MaClasse();
MaClasse m2 = m1.clone();
```

m1 et m2 ne contiennent plus la même référence et pointent donc sur des objets différents.

4.2.4. Les références et la comparaison d'objets

Les variables de type objet que l'on déclare ne contiennent pas un objet mais une référence vers cet objet. Lorsque l'on écrit c1 = c2 (c1 et c2 sont des objets), on copie la référence de l'objet c2 dans c1 : c1 et c2 réfèrent au même objet (ils pointent sur le même objet). L'opérateur == compare ces références. Deux objets avec des propriétés identiques sont deux objets distincts :

Exemple :

```
Rectangle r1 = new Rectangle(100,50);
Rectangle r2 = new Rectangle(100,50);
if (r1 == r1) { ... } // vrai
if (r1 == r2) { ... } // faux
```

Pour comparer l'égalité des variables de deux instances, il faut munir la classe d'une méthode à cet effet : la méthode equals() héritée de Object.

Pour s'assurer que deux objets sont de la même classe, il faut utiliser la méthode getClass() de la classe Object dont toutes les classes héritent.

Exemple :

```
(obj1.getClass().equals(obj2.getClass())
```

4.2.5. L'objet null

L'objet null est utilisable partout. Il n'appartient pas à une classe mais il peut être utilisé à la place d'un objet de n'importe quelle classe ou comme paramètre. null ne peut pas être utilisé comme un objet normal : il n'y a pas d'appel de méthodes et aucune classe ne peut en hériter.

Le fait d'initialiser une variable référant un objet à null permet au ramasse miette de libérer la mémoire allouée à l'objet.

4.2.6. Les variables de classes

Elles ne sont définies qu'une seule fois quel que soit le nombre d'objets instanciés de la classe. Leur déclaration est accompagnée du mot clé static

Exemple :

```
public class MaClasse() {
    static int compteur = 0;
}
```

L'appartenance des variables de classe à une classe entière et non à un objet spécifique permet de remplacer le nom de la variable par le nom de la classe.

Exemple :

```
MaClasse m = new MaClasse();
int c1 = m.compteur;
int c2 = MaClasse.compteur;
// c1 et c2 possèdent la même valeur.
```

Ce type de variable est utile pour par exemple compter le nombre d'instanciation de la classe qui est faite.

4.2.7. La variable this

Cette variable sert à référencer dans une méthode l'instance de l'objet en cours d'utilisation. this est un objet qui est égale à l'instance de l'objet dans lequel il est utilisé.

Exemple :

```
private int nombre;
public maclasse(int nombre) {
   nombre = nombre; // variable de classe = variable en paramètre du constructeur
}
```

Il est préférable de préfixer la variable d'instance par le mot clé this.

Exemple :

```
this.nombre = nombre;
```

Cette référence est habituellement implicite :

Exemple :

```
class MaClasse() {
  String chaine = " test " ;
  public String getChaine() { return chaine; }
  // est équivalent à public String getChaine() { return this.chaine; }
}
```

This est aussi utilisé quand l'objet doit appeler une méthode en se passant lui même en paramètre de l'appel.

4.2.8. L'opérateur instanceof

L'opérateur instanceof permet de déterminer la classe de l'objet qui lui est passé en paramètre. La syntaxe est objet instanceof classe

Exemple :

```
void testClasse(Object o) {
   if (o instanceof MaClasse )
      System.out.println(" o est une instance de la classe MaClasse ");
   else System.out.println(" o n'est pas un objet de la classe MaClasse ");
}
```

Il n'est toutefois pas possible d'appeler une méthode de l'objet car il est passé en paramètre avec un type Object

Exemple :

```
void afficheChaine(Object o) {
    if (o instanceof MaClasse)
        System.out.println(o.getChaine());
        // erreur à la compil car la méthode getChaine()
        //n'est pas définie dans la classe Object
}
```

Pour résoudre le problème, il faut utiliser la technique du casting (conversion).

Exemple :

```
void afficheChaine(Object o) {
   if (o instanceof MaClasse)
   {
      MaClasse m = (MaClasse) o;
      System.out.println(m.getChaine());
      // OU System.out.println( ((MaClasse) o).getChaine() );
   }
}
```

4.3. Les modificateurs d'accès

Ils se placent avant ou après le type de l'objet mais la convention veut qu'ils soient placés avant.

Ils s'appliquent aux classes et/ou aux méthodes et/ou aux attributs.

Ils ne peuvent pas être utilisés pour qualifier des variables locales : seules les variables d'instances et de classes peuvent en profiter.

Ils assurent le contrôle des conditions d'héritage, d'accès aux éléments et de modification de données par les autres objets.

4.3.1. Les mots clés qui gèrent la visibilité des entités

De nombreux langages orientés objet introduisent des attributs de visibilité pour réglémenter l'accès aux classes et aux objets, aux méthodes et aux données.

Il existe 3 modificateurs qui peuvent être utilisés pour définir les attributs de visibilité des entités (classes, méthodes ou attributs) : public, private et protected. Leur utilisation permet de définir des niveaux de protection différents (présentés dans un ordre croissant de niveau de protection offert) :

Modificateur	Rôle
public	Une variable, méthode ou classe déclarée public est visible par tous les autres objets. Dans la version 1.0, une seule classe public est permise par fichier et son nom doit correspondre à celui du fichier. Dans la philosophie orientée objet aucune donnée d'une classe ne devrait être déclarée publique : il est préférable d'écrire des méthodes pour la consulter et la modifier
par défaut : package friendly	Il n'existe pas de mot clé pour définir ce niveau, qui est le niveau par défaut lorsqu'aucun modificateur n'est précisé. Cette déclaration permet à une entité (classe, méthode ou variable) d'être visible par toutes les classes se trouvant dans le même package.
protected	Si une méthode ou une variable est déclarée protected , seules les méthodes présentes dans le même package que cette classe ou ses sous classes pourront y accéder. On ne peut pas qualifier une classe avec protected.
private	C'est le niveau de protection le plus fort. Les composants ne sont visibles qu'à l'intérieur de la classe : ils ne peuvent être modifiés que par des méthodes définies dans la classe prévues à cet effet. Les méthodes déclarées private ne peuvent pas être en même temps déclarée abstract car elles ne peuvent pas être redéfinies dans les classes filles.

Ces modificateurs d'accès sont mutuellement exclusifs.

4.3.2. Le mot clé static

Le mot clé static s'applique aux variables et aux méthodes.

Les variables d'instance sont des variables propres à un objet. Il est possible de définir une variable de classe qui est partagée entre toutes les instances d'une même classe : elle n'existe donc qu'une seule fois en mémoire. Une telle variable permet de stocker une constante ou une valeur modifiée tour à tour par les instances de la classe. Elle se définit avec le mot clé static.

Exemple :

```
public class Cercle {
    static float pi = 3.1416f;
    float rayon;
    public Cercle(float rayon) { this.rayon = rayon; }
    public float surface() { return rayon * rayon * pi;}
}
```

Il est aussi possible par exemple de mémoriser les valeurs min et max d'un ensemble d'objets de même classe.

Une méthode static est une méthode qui n'agit pas sur des variables d'instance mais uniquement sur des variables de classe. Ces méthodes peuvent être utilisées sans instancier un objet de la classe. Les méthodes ainsi définies peuvent être appelées avec la notation classe.methode() au lieu de objet.methode() : la première forme est fortement recommandée pour éviter toute confusion.

Il n'est pas possible d'appeler une méthode d'instance ou d'accéder à une variable d'instance à partir d'une méthode de classe statique.

4.3.3. Le mot clé final

Le mot clé final s'applique aux variables de classe ou d'instance ou locales, aux méthodes, aux paramètres d'une méthode et aux classes. Il permet de rendre l'entité sur laquelle il s'applique non modifiable une fois qu'elle est déclarée pour une méthode ou une classe et initialisée pour une variable.

Une variable qualifiée de final signifie que la valeur de la variable ne peut plus être modifiée une fois que celle ci est initialisée.

Exemple :

```
package com.jmdoudoux.test;

public class Constante2 {

  public final int constante;

  public Constante2() {
    this.constante = 10;
  }

}
```

Une fois la variable déclarée final initialisée, il n'est plus possible de modifier sa valeur. Une vérification est opérée par le compilateur.

Exemple :

```
package com.jmdoudoux.test;

public class Constante1 {

  public static final int constante = 0;

  public Constante1() {
    this.constante = 10;
  }

}
```

Résultat :

```
C:\>javac Constante1.java
Constante1.java:6: cannot assign a value to final variable constante
    this.constante = 10;
    ^
1 error
```

Les constantes sont qualifiées avec les modificateurs final et static.

Exemple :

```
public static final float PI = 3.141f;
```

Une méthode déclarée final ne peut pas être redéfinie dans une sous classe. Une méthode possédant le modificateur final pourra être optimisée par le compilateur car il est garanti qu'elle ne sera pas sous classée.

Lorsque le modificateur final est ajouté à une classe, il est interdit de créer une classe qui en hérite.

Pour une méthode ou une classe, on renonce à l'héritage mais ceci peut s'avérer nécessaire pour des questions de sécurité ou de performance. Le test de validité de l'appel d'une méthode est bien souvent repoussé à l'exécution, en fonction du type de l'objet appelé (c'est la notion de polymorphisme qui sera détaillée ultérieurement). Ces tests ont un coût en termes de performance.

4.3.4. Le mot clé abstract

Le mot clé abstract s'applique aux méthodes et aux classes.

Abstract indique que la classe ne pourra être instanciée telle quelle. De plus, toutes les méthodes de cette classe abstract ne sont pas implémentées et devront être redéfinies par des méthodes complètes dans ses sous classes.

Abstract permet de créer une classe qui sera une sorte de moule. Toutes les classes dérivées pourront profiter des méthodes héritées et n'auront à implémenter que les méthodes déclarées abstract.

Exemple :

```
abstract class ClasseAbstraite {
    ClasseBastraite() { ... //code du constructeur }
    void methode() { ... // code partagé par tous les descendants}
    abstract void methodeAbstraite();
}

class ClasseComplete extends ClasseAbstraite {
    ClasseComplete() { super(); ... }
    void methodeAbstraite() { ... // code de la méthode }
    // void methode est héritée
}
```

Une méthode abstraite est une méthode déclarée avec le modificateur abstract et sans corps. Elle correspond à une méthode dont on veut forcer l'implémentation dans une sous classe. L'abstraction permet une validation du codage : une sous classe sans le modificateur abstract et sans définition explicite d'une ou des méthodes abstraites génère une erreur de compilation.

Une classe est automatiquement abstraite dès qu'une de ses méthodes est déclarée abstraite. Il est possible de définir une classe abstraite sans méthodes abstraites.

4.3.5. Le mot clé synchronized

Permet de gérer l'accès concurrent aux variables et méthodes lors de traitement de thread (exécution « simultanée » de plusieurs petites parties de code du programme)

4.3.6. Le mot clé volatile

Le mot clé volatile s'applique aux variables.

Précise que la variable peut être changée par un périphérique ou de manière asynchrone. Cela indique au compilateur de ne pas stocker cette variable dans des registres. A chaque utilisation, on lit la valeur et on réécrit immédiatement le résultat s'il a changé.

4.3.7. Le mot clé native

Une méthode native est une méthode qui est implémentée dans un autre langage. L'utilisation de ce type de méthode limite la portabilité du code mais permet une vitesse exécution plus rapide.

4.4. Les propriétés ou attributs

Les données d'une classe sont contenues dans des variables nommées propriétés ou attributs. Ce sont des variables qui peuvent être des variables d'instances, des variables de classes ou des constantes.

4.4.1. Les variables d'instances

Une variable d'instance nécessite simplement une déclaration de la variable dans le corps de la classe.

Exemple :

```
public class MaClasse {
   public int valeur1 ;
   int valeur2 ;
   protected int valeur3 ;
   private int valeur4 ;
}
```

Chaque instance de la classe a accès à sa propre occurrence de la variable.

4.4.2. Les variables de classes

Les variables de classes sont définies avec le mot clé static

Exemple (code Java 1.1) :

```
public class MaClasse {
   static int compteur ;
}
```

Chaque instance de la classe partage la même variable.

4.4.3. Les constantes

Les constantes sont définies avec le mot clé final : leur valeur ne peut pas être modifiée une fois qu'elles sont initialisées.

Exemple (code Java 1.1) :

```
public class MaClasse {
   final double pi=3.14 ;
}
```

4.5. Les méthodes

Les méthodes sont des fonctions qui implémentent les traitements de la classe.

4.5.1. La syntaxe de la déclaration

La syntaxe de la déclaration d'une méthode est :

modificateurs type_retourné nom_méthode (arg1, ...) {... } // définition des variables locales et du bloc d'instructions }

Le type retourné peut être élémentaire ou correspondre à un objet. Si la méthode ne retourne rien, alors on utilise void.

Le type et le nombre d'arguments déclarés doivent correspondre au type et au nombre d'arguments transmis. Il n'est pas possible d'indiquer des valeurs par défaut dans les paramètres. Les arguments sont passés par valeur : la méthode fait une copie de la variable qui lui est locale. Lorsqu'un objet est transmis comme argument à une méthode, cette dernière reçoit une référence qui désigne son emplacement mémoire d'origine et qui est une copie de la variable. Il est possible de modifier l'objet grâce à ces méthodes mais il n'est pas possible de remplacer la référence contenue dans la variable passée en paramètre : ce changement n'aura lieu que localement à la méthode.

Les modificateurs de méthodes sont :

Modificateur	Rôle
public	la méthode est accessible aux méthodes des autres classes
private	l'usage de la méthode est réservé aux autres méthodes de la même classe
protected	la méthode ne peut être invoquée que par des méthodes de la classe ou de ses sous classes
final	la méthode ne peut être modifiée (redéfinition lors de l'héritage interdite)
static	la méthode appartient simultanément à tous les objets de la classe (comme une constante déclarée à l'intérieur de la classe). Il est inutile d'instancier la classe pour appeler la méthode mais la méthode ne peut pas manipuler de variable d'instance. Elle ne peut utiliser que des variables de classes.
synchronized	la méthode fait partie d'un thread. Lorsqu'elle est appelée, elle barre l'accès à son instance. L'instance est à nouveau libérée à la fin de son exécution.
native	le code source de la méthode est écrit dans un autre langage

Sans modificateur, la méthode peut être appelée par toutes autres méthodes des classes du package auquel appartient la classe.

La valeur de retour de la méthode doit être transmise par l'instruction return. Elle indique la valeur que prend la méthode et termine celle ci : toutes les instructions qui suivent return sont donc ignorées.

Exemple :

```
int add(int a, int b) {
   return a + b;
}
```

Il est possible d'inclure une instruction return dans une méthode de type void : cela permet de quitter la méthode.

La méthode main() de la classe principale d'une application doit être déclarée de la façon suivante : public static void main (String args[]) { ... }

Exemple :

```
public class MonApp1 {

  public static void main(String[] args) {
    System.out.println("Bonjour");
  }
}
```

Cette déclaration de la méthode main() est imposée par la machine virtuelle pour reconnaitre le point d'entrée d'une application. Si la déclaration de la méthode main() diffère, une exception sera levée lors de la tentative d'exécution par la machine virtuelle.

Exemple :

```
public class MonApp2 {

  public static int main(String[] args) {
    System.out.println("Bonjour");
    return 0;
  }
}
```

Résultat :

```
C:\>javac MonApp2.java

C:\>java MonApp2
Exception in thread "main" java.lang.NoSuchMethodError: main
```

Si la méthode retourne un tableau alors les caractères [] peuvent être précisés après le type de retour ou après la liste des paramètres :

Exemple :

```
int[] valeurs() { ... }
int valeurs()[] { ... }
```

4.5.2. La transmission de paramètres

Lorsqu'un objet est passé en paramètre, ce n'est pas l'objet lui même qui est passé mais une référence sur l'objet. La référence est bien transmise par valeur et ne peut pas être modifiée mais l'objet peut être modifié via un message (appel d'une méthode).

Pour transmettre des arguments par référence à une méthode, il faut les encapsuler dans un objet qui prévoit les méthodes nécessaires pour les mises à jour.

Si un objet o transmet sa variable d'instance v en paramètre à une méthode m, deux situations sont possibles :

- si v est une variable primitive alors elle est passée par valeur : il est impossible de la modifier dans m pour que v en retour contienne cette nouvelle valeur.
- si v est un objet alors m pourra modifier l'objet en utilisant une méthode de l'objet passé en paramètre.

4.5.3. L'émission de messages

Un message est émis lorsqu'on demande à un objet d'exécuter l'une de ses méthodes.

La syntaxe d'appel d'une méthode est : nom_objet.nom_méthode(parametre, ...) ;

Si la méthode appelée ne contient aucun paramètre, il faut laisser les parenthèses vides.

4.5.4. L'enchainement de références à des variables et à des méthodes

Exemple :

```
System.out.println("bonjour");
```

Deux classes sont impliquées dans l'instruction : System et PrintStream. La classe System possède une variable nommée out qui est un objet de type PrintStream. Println() est une méthode de la classe PrintStream. L'instruction signifie : « utilise la méthode println() de la variable out de la classe System ».

4.5.5. La surcharge de méthodes

La surcharge d'une méthode permet de définir plusieurs fois une même méthode avec des arguments différents. Le compilateur choisi la méthode qui doit être appelée en fonction du nombre et du type des arguments. Ceci permet de simplifier l'interface des classes vis à vis des autres classes.

Une méthode est surchargée lorsqu'elle exécute des actions différentes selon le type et le nombre de paramètres transmis.

Il est donc possible de donner le même nom à deux méthodes différentes à condition que les signatures de ces deux méthodes soient différentes. La signature d'une méthode comprend le nom de la classe, le nom de la méthode et les types des paramètres.

Exemple :

```
class affiche{
    public void afficheValeur(int i) {
        System.out.println(" nombre entier = " + i);
    }

    public void afficheValeur(float f) {
        System.out.println(" nombre flottant = " + f);
    }
}
```

Il n'est pas possible d'avoir deux méthodes de même nom dont tous les paramètres sont identiques et dont seul le type retourné diffère.

Exemple :

```
class Affiche{

    public float convert(int i){
         return((float) i);
    }

    public double convert(int i){
        return((double) i);
    }
```

```
}
```

Résultat :

```
C:\>javac Affiche.java
Affiche.java:5: Methods can't be redefined with a different return type: double
convert(int) was float convert(int)
public double convert(int i){
       ^
1 error
```

4.5.6. Les constructeurs

La déclaration d'un objet est suivie d'une sorte d'initialisation par le moyen d'une méthode particulière appelée constructeur pour que les variables aient une valeur de départ. Elle n'est systématiquement invoquée que lors de la création d'un objet.

Le constructeur suit la définition des autres méthodes excepté que son nom doit obligatoirement correspondre à celui de la classe et qu'il n'est pas typé, pas même void, donc il ne peut pas y avoir d'instruction return dans un constructeur. On peut surcharger un constructeur.

La définition d'un constructeur est facultative. Si elle n'est pas définie, la machine virtuelle appelle un constructeur par défaut vide créé automatiquement. Dès qu'un constructeur est explicitement défini, Java considère que le programmeur prend en charge la création des constructeurs et que le mécanisme par défaut, qui correspond à un constructeur sans paramètres, est supprimé. Si on souhaite maintenir ce mécanisme, il faut définir explicitement un constructeur sans paramètres.

Il existe plusieurs manières de définir un constructeur :

1. le constructeur simple : ce type de constructeur ne nécessite pas de définition explicite : son existence découle automatiquement de la définition de la classe.

   Exemple :

   ```
   public MaClasse() {}
   ```

2. le constructeur avec initialisation fixe : il permet de créer un constructeur par défaut

   Exemple :

   ```
   public MaClasse() {
       nombre = 5;
   }
   ```

3. le constructeur avec initialisation des variables : pour spécifier les valeurs de données à initialiser on peut les passer en paramètres au constructeur

   Exemple :

   ```
   public MaClasse(int valeur) {
        nombre = valeur;
   }
   ```

4.5.7. Le destructeur

Un destructeur permet d'exécuter du code lors de la libération, par le garbage collector, de l'espace mémoire occupé par l'objet. En Java, les destructeurs appelés finaliseurs (finalizers), sont automatiquement invoqués par le garbage collector.

Pour créer un finaliseur, il faut redéfinir la méthode finalize() héritée de la classe Object.

Attention : selon l'implémentation du garbage collector dans la machine virtuelle, il n'est pas possible de prévoir le moment ou un objet sera traité par le garbage collector. De plus, l'appel du finaliseur n'est pas garanti : par exemple, si la machine virtuelle est brusquement arrêtée par l'utilisateur, le ramasse miette ne libérera pas la mémoire des objets en cours d'utilisation et les finaliseurs de ces objets ne seront pas appelés.

4.5.8. Les accesseurs

L'encapsulation permet de sécuriser l'accès aux données d'une classe. Ainsi, les données déclarées private à l'intérieur d'une classe ne peuvent être accédées et modifiées que par des méthodes définies dans la même classe. Si une autre classe veut accéder aux données de la classe, l'opération n'est possible que par l'intermédiaire d'une méthode de la classe prévue à cet effet. Ces appels de méthodes sont appelés « échanges de message ».

Un accesseur est une méthode publique qui donne l'accès à une variable d'instance privée. Pour une variable d'instance, il peut ne pas y avoir d'accesseur, un seul accesseur en lecture ou un accesseur en lecture et un autre en écriture. Par convention, les accesseurs en lecture commencent par get et les accesseurs en écriture commencent par set.

Exemple :

```
private int valeur = 13;

public int getValeur(){
    return(valeur);
}

public void setValeur(int val) {
    valeur = val;
}
```

Pour un attribut de type booléen, il est possible de faire commencer l'accesseur en lecture par is au lieu de get.

4.6. L'héritage

L'héritage est un mécanisme qui facilite la réutilisation du code et la gestion de son évolution. Elle définit une relation entre deux classes :

- une classe mère ou super classe
- une classe fille ou sous classe qui hérite de sa classe mère

4.6.1. Le principe de l'héritage

Grâce à l'héritage, les objets d'une classe fille ont accès aux données et aux méthodes de la classe parent et peuvent les étendre. Les sous classes peuvent redéfinir les variables et les méthodes héritées. Pour les variables, il suffit de les redéclarer sous le même nom avec un type différent. Les méthodes sont redéfinies avec le même nom, les mêmes types et le même nombre d'arguments, sinon il s'agit d'une surcharge.

L'héritage successif de classes permet de définir une hiérarchie de classe qui se compose de super classes et de sous classes. Une classe qui hérite d'une autre est une sous classe et celle dont elle hérite est une super classe. Une classe peut avoir plusieurs sous classes. Une classe ne peut avoir qu'une seule classe mère : il n'y a pas d'héritage multiple en Java.

Object est la classe parente de toutes les classes en Java. Toutes les variables et méthodes contenues dans Object sont accessibles à partir de n'importe quelle classe car par héritages successifs toutes les classes héritent d'Object.

4.6.2. La mise en oeuvre de l'héritage

On utilise le mot clé extends pour indiquer qu'une classe hérite d'une autre. En l'absence de ce mot réservé associé à une classe, le compilateur considère la classe Object comme classe parent.

Exemple :

```
class Fille extends Mere { ... }
```

Pour invoquer une méthode d'une classe parent, il suffit d'indiquer la méthode préfixée par super. Pour appeler le constructeur de la classe parent il suffit d'écrire super(paramètres) avec les paramètres adéquats.

Le lien entre une classe fille et une classe parent est géré par le langage : une évolution des règles de gestion de la classe parent conduit à modifier automatiquement la classe fille dès que cette dernière est recompilée.

En Java, il est obligatoire dans un constructeur d'une classe fille de faire appel explicitement ou implicitement au constructeur de la classe mère.

4.6.3. L'accès aux propriétés héritées

Les variables et méthodes définies avec le modificateur d'accès public restent publiques à travers l'héritage et toutes les autres classes.

Une variable d'instance définie avec le modificateur private est bien héritée mais elle n'est pas accessible directement mais via les méthodes héritées.

Si l'on veut conserver pour une variable d'instance une protection semblable à celle assurée par le modificateur private, il faut utiliser le modificateur protected. La variable ainsi définie sera héritée dans toutes les classes descendantes qui pourront y accéder librement mais ne sera pas accessible hors de ces classes directement.

4.6.4. La redéfinition d'une méthode héritée

La redéfinition d'une méthode héritée doit impérativement conserver la déclaration de la méthode parent (type et nombre de paramètres, la valeur de retour et les exceptions propagées doivent être identique).

Si la signature de la méthode change, ce n'est plus une redéfinition mais une surcharge. Cette nouvelle méthode n'est pas héritée : la classe mère ne possède pas de méthode possédant cette signature.

4.6.5. Le polymorphisme

Le polymorphisme est la capacité, pour un même message de correspondre à plusieurs formes de traitements selon l'objet auquel ce message est adressé. La gestion du polymorphisme est assurée par la machine virtuelle dynamiquement à l'exécution.

4.6.6. Le transtypage induit par l'héritage facilite le polymorphisme

L'héritage définit un cast implicite de la classe fille vers la classe mère : on peut affecter à une référence d'une classe n'importe quel objet d'une de ses sous classes.

Exemple : la classe Employe hérite de la classe Personne

```
    Personne p = new Personne ("Dupond", "Jean");
    Employe e = new Employe("Durand", "Julien", 10000);
    p = e ; // ok : Employe est une sous classe de Personne
    Objet obj;
    obj = e ; // ok : Employe herite de Personne qui elle même hérite de Object
```

Il est possible d'écrire le code suivant si Employe hérite de Personne

Exemple :

```
    Personne[] tab = new Personne[10];
    tab[0] = new Personne("Dupond","Jean");
    tab[1] = new Employe("Durand", "Julien", 10000);
```

Il est possible de surcharger une méthode héritée : la forme de la méthode à exécuter est choisie en fonction des paramètres associés à l'appel.

Compte tenu du principe de l'héritage, le temps d'exécution du programme et la taille du code source et de l'exécutable augmentent.

4.6.7. Les interfaces et l'héritage multiple

Avec l'héritage multiple, une classe peut hériter en même temps de plusieurs super classes. Ce mécanisme n'existe pas en Java. Les interfaces permettent de mettre en oeuvre un mécanisme de remplacement.

Une interface est un ensemble de constantes et de déclarations de méthodes correspondant un peu à une classe abstraite. C'est une sorte de standard auquel une classe peut répondre. Tous les objets qui se conforment à cette interface (qui implémentent cette interface) possèdent les méthodes et les constantes déclaréees dans celle-ci. Plusieurs interfaces peuvent être implémentées dans une même classe.

Les interfaces se déclarent avec le mot clé interface et sont intégrées aux autres classes avec le mot clé implements. Une interface est implicitement déclarée avec le modificateur abstract.

Déclaration d'une interface :

```
[public] interface nomInterface [extends nomInterface1, nomInterface2 ... ] {
    // insérer ici des méthodes ou des champs static
}
```

Implémentation d'une interface :

```
Modificateurs class nomClasse [extends superClasse]
   [implements nomInterface1, nomInterface 2, ...] {
    //insérer ici des méthodes et des champs

}
```

Exemple :

```
interface AfficheType {
    void afficherType();
}

class Personne implements AfficheType {

    public void afficherType() {
        System.out.println(" Je suis une personne ");
    }
}

class Voiture implements AfficheType {

    public void afficherType() {

        System.out.println(" Je suis une voiture ");
    }
}
```

Exemple : déclaration d'une interface à laquelle doit se conformer tout individus

```
interface Individu {
    String getNom();
    String getPrenom();
    Date getDateNaiss();
}
```

Toutes les méthodes d'une interface sont abstraites : elles sont implicitement déclarées comme telles.

Une interface peut être d'accès public ou package. Si elle est publique, toutes ses méthodes sont implicitement publiques même si elles ne sont pas déclarées avec le modificateur public. Si elle est d'accès package, il s'agit d'une interface d'implémentation pour les autres classes du package et ses méthodes ont le même accès package : elles sont accessibles à toutes les classes du packages.

Les seules variables que l'on peut définir dans une interface sont des variables de classe qui doivent être constantes : elles sont donc implicitement déclarées avec le modificateur static et final même si elles sont définies avec d'autres modificateurs.

Exemple :

```
public interface MonInterface {
  public int VALEUR=0;
  void maMethode();
}
```

Toute classe qui implémente cette interface doit au moins posséder les méthodes qui sont déclarées dans l'interface. L'interface ne fait que donner une liste de méthodes qui seront à définir dans les classes qui implémentent l'interface.

Les méthodes déclarées dans une interface publique sont implicitement publiques et elles sont héritées par toutes les classes qui implémentent cette interface. Une telle classe doit, pour être instanciable, définir toutes les méthodes héritées de l'interface.

Une classe peut implémenter une ou plusieurs interfaces tout en héritant de sa classe mère.

L'implémentation d'une interface définit un cast : l'implémentation d'une interface est une forme d'héritage. Comme pour l'héritage d'une classe, l'héritage d'une classe qui implémente une interface définit un cast implicite de la classe fille vers cette interface. Il est important de noter que dans ce cas il n'est possible de faire des appels qu'à des méthodes de l'interface. Pour utiliser des méthodes de l'objet, il faut définir un cast explicite : il est préférable de contrôler la classe de l'objet pour éviter une exception ClassCastException à l'exécution

4.6.8. Des conseils sur l'héritage

Lors de la création d'une classe « mère » il faut tenir compte des points suivants :

- la définition des accès aux variables d'instances, très souvent privées, doit être réfléchie entre protected et private
- pour empêcher la redéfinition d'une méthode (surcharge) il faut la déclarer avec le modificateur final

Lors de la création d'une classe fille, pour chaque méthode héritée qui n'est pas final, il faut envisager les cas suivants :

- la méthode héritée convient à la classe fille : on ne doit pas la redéfinir
- la méthode héritée convient mais partiellement du fait de la spécialisation apportée par la classe fille : il faut la redéfinir voir la surcharger. La plupart du temps une redéfinition commencera par appeler la méthode héritée (via super) pour garantir l'évolution du code
- la méthode héritée ne convient pas : il faut redéfinir ou surcharger la méthode sans appeler la méthode héritée lors de la redéfinition.

4.7. Les packages

4.7.1. La définition d'un package

En Java, il existe un moyen de regrouper des classes voisines ou qui couvrent un même domaine : ce sont les packages. Pour réaliser un package, on écrit un nombre quelconque de classes dans plusieurs fichiers d'un même répertoire et au début de chaque fichier on met la directive ci dessous ou nom-du-package doit être identique au nom du répertoire :

package nomPackage;

La hiérarchie d'un package se retrouve dans l'arborescence du disque dur puisque chaque package est dans un répertoire nommé du nom du package.

Remarque : Il est préférable de laisser les fichiers source .java avec les fichiers compilés .class

D'une façon générale, l'instruction package associe toutes les classes qui sont définies dans un fichier source à un même package.

Le mot clé package doit être la première instruction dans un fichier source et il ne doit être présent qu'une seule fois dans le fichier source (une classe ne peut pas appartenir à plusieurs packages).

4.7.2. L'utilisation d'un package

Pour utiliser ensuite le package ainsi créé, on l'importe dans le fichier :

import nomPackage.*;

Pour importer un package, il y a deux méthodes si le chemin de recherche est correctement renseigné : préciser un nom de classe ou interface qui sera l'unique entité importé ou mettre une * indiquant toutes les classes et interfaces définies dans le package.

Exemple	Rôle
`import nomPackage.*;`	toutes les classes du package sont importées

`import nomPackage.nomClasse;`	appel à une seule classe : l'avantage de cette notation est de réduire le temps de compilation

Attention : l'astérisque n'importe pas les sous paquetages. Par exemple, il n'est pas possible d'écrire import java.*.

Il est possible d'appeler une méthode d'un package sans inclure ce dernier dans l'application en précisant son nom complet :

nomPackage.nomClasse.nomméthode(arg1, arg2 ...)

Il existe plusieurs types de packages : le package par défaut (identifié par le point qui représente le répertoire courant et permet de localiser les classes qui ne sont pas associées à un package particulier), les packages standards qui sont empaquetés dans le fichier classes.zip et les packages personnels

Le compilateur implémente automatiquement une commande import lors de la compilation d'un programme Java même si elle ne figure pas explicitement au début du programme : import java.lang.*; Ce package contient entre autre les classes de base de tous les objets Java dont la classe Object.

Un package par défaut est systématiquement attribué par le compilateur aux classes qui sont définies sans déclarer explicitement une appartenance à un package. Ce package par défaut correspond au répertoire courant qui est le répertoire de travail.

4.7.3. La collision de classes

Deux classes entre en collision lorsqu'elles portent le même nom mais qu'elles sont définies dans des packages différents. Dans ce cas, il faut qualifier explicitement le nom de la classe avec le nom complet du package.

4.7.4. Les packages et l'environnement système

Les classes Java sont importées par le compilateur (au moment de la compilation) et par la machine virtuelle (au moment de l'exécution). Les techniques de chargement des classes varient en fonction de l'implémentation de la machine virtuelle. Dans la plupart des cas, une variable d'environnement CLASSPATH référence tous les répertoires qui hébergent des packages susceptibles d'être importés.

Exemple sous Windows :

CLASSPATH = .;C:\Java\JDK\Lib\classes.zip; C:\rea_java\package

L'importation des packages ne fonctionne que si le chemin de recherche spécifié dans une variable particulière pointe sur les packages, sinon le nom du package devra refléter la structure du répertoire ou il se trouve. Pour déterminer l'endroit où se trouvent les fichiers .class à importer, le compilateur utilise une variable d'environnement dénommée CLASSPATH. Le compilateur peut lire les fichiers .class comme des fichiers indépendants ou comme des fichiers ZIP dans lesquels les classes sont réunies et compressées.

4.8. Les classes internes

Les classes internes (inner classes) sont une extension du langage Java introduite dans la version 1.1 du JDK. Ce sont des classes qui sont définies dans une autre classe. Les difficultés dans leur utilisation concernent leur visibilité et leur accès aux membres de la classe dans laquelle elles sont définies.

Exemple très simple :

```
public class ClassePrincipale1 {
  class ClasseInterne {
  }
}
```

Les classes internes sont particulièrement utiles pour :

- permettre de définir une classe à l'endroit ou une seule autre en a besoin
- définir des classes de type adapter (essentiellement à partir du JDK 1.1 pour traiter des événements émis par les interfaces graphiques)
- définir des méthodes de type callback d'une façon générale

Pour permettre de garder une compatibilité avec la version précédente de la JVM, seul le compilateur a été modifié. Le compilateur interprète la syntaxe des classes internes pour modifier le code source et générer du byte code compatible avec la première JVM.

Il est possible d'imbriquer plusieurs classes internes. Java ne possèdent pas de restrictions sur le nombre de classes qu'il est ainsi possible d'imbriquer. En revanche une limitation peut intervenir au niveau du système d'exploitation en ce qui concerne la longueur du nom du fichier .class généré pour les différentes classes internes.

Si plusieurs classes internes sont imbriquées, il n'est pas possible d'utiliser un nom pour la classe qui soit déjà attribuée à une de ces classes englobantes. Le compilateur génèrera une erreur à la compilation.

Exemple :

```
public class ClassePrincipale6 {
  class ClasseInterne1 {
    class ClasseInterne2 {
      class ClasseInterne3 {
      }
    }
  }
}
```

Le nom de la classe interne utilise la notation qualifiée avec le point préfixé par le nom de la classe principale. Ainsi, pour utiliser ou accéder à une classe interne dans le code, il faut la préfixer par le nom de la classe principale suivi d'un point.

Cependant cette notation ne représente pas physiquement le nom du fichier qui contient le byte code. Le nom du fichier qui contient le byte code de la classe interne est modifié par le compilateur pour éviter des conflits avec d'autres noms d'entité : à partir de la classe principale, les points de séparation entre chaque classe interne sont remplacé par un caractère $ (dollar).

Par exemple, la compilation du code de l'exemple précédent génère quatre fichiers contenant le byte code :
ClassePrincipale6$ClasseInterne1$ClasseInterne2ClasseInterne3.class
ClassePrincipale6$ClasseInterne1$ClasseInterne2.class
ClassePrincipale6$ClasseInterne1.class
ClassePrincipale6.class

L'utilisation du signe $ entre la classe principale et la classe interne permet d'éviter des confusions de nom entre le nom d'une classe appartenant à un package et le nom d'une classe interne.

L'avantage de cette notation est de créer un nouvel espace de nommage qui dépend de la classe et pas d'un package. Ceci renforce le lien entre la classe interne et sa classe englobante.

C'est le nom du fichier qu'il faut préciser lorsque l'on tente de charger la classe avec la méthode forName() de la classe Class. C'est aussi sous cette forme qu'est restitué le résultat d'un appel aux méthodes getClass().getName() sur un objet qui est une classe interne.

Exemple :

```
public class ClassePrincipale8 {
  public class ClasseInterne {
  }

  public static void main(String[] args) {
    ClassePrincipale8 cp = new ClassePrincipale8();
    ClassePrincipale8.ClasseInterne ci = cp. new ClasseInterne() ;
    System.out.println(ci.getClass().getName());
  }
}
```

Résultat :

```
java ClassePrincipale8
ClassePrincipale8$ClasseInterne
```

L'accessibilité à la classe interne respecte les règles de visibilité du langage. Il est même possible de définir une classe interne private pour limiter son accès à sa seule classe principale.

Exemple :

```
public class ClassePrincipale7 {
        private class ClasseInterne {
        }
}
```

Il n'est pas possible de déclarer des membres statiques dans une classe interne :

Exemple :

```
public class ClassePrincipale10 {
        public class ClasseInterne {
                static int var = 3;
        }
}
```

Résultat :

```
javac ClassePrincipale10.java
ClassePrincipale10.java:3: Variable var can't be static in inner class ClassePri
ncipale10. ClasseInterne.  Only members of interfaces and top-level classes can
be static.
                static int var = 3;
                               ^
1 error
```

Pour pouvoir utiliser une variable de classe dans une classe interne, il faut la déclarer dans sa classe englobante.

Il existe quatre types de classes internes :

- les classes internes non statiques : elles sont membres à part entière de la classe qui les englobe et peuvent accéder à tous les membres de cette dernière
- les classes internes locales : elles sont définies dans un block de code. Elles peuvent être static ou non.
- les classes internes anonymes : elles sont définies et instanciées à la volée sans posséder de nom
- les classe internes statiques : elles sont membres à part entière de la classe qui les englobe et peuvent accéder uniquement aux membres statiques de cette dernière

4.8.1. Les classes internes non statiques

Les classes internes non statiques (member inner-classes) sont définies dans une classe dite « principale » (top-level class) en tant que membre de cette classe. Leur avantage est de pouvoir accéder aux autres membres de la classe principale même ceux déclarés avec le modificateur private.

Exemple :

```
public class ClassePrincipale20 {
        private int valeur = 1;

        class ClasseInterne {
                public void afficherValeur() {
                        System.out.println("valeur = "+valeur);
                }
        }

        public static void main(String[] args) {
                ClassePrincipale20 cp = new ClassePrincipale20();
                ClasseInterne ci = cp. new ClasseInterne();
                ci.afficherValeur();
        }
}
```

Résultat :

```
C:\testinterne>javac ClassePrincipale20.java

C:\testinterne>java ClassePrincipale20
valeur = 1
```

Le mot clé this fait toujours référence à l'instance en cours. Ainsi this.var fait référence à la variable var de l'instance courante. L'utilisation du mot clé this dans une classe interne fait donc référence à l'instance courante de cette classe interne.

Exemple :

```
public class ClassePrincipale16 {
        class ClasseInterne {
                int var = 3;

                public void affiche() {
                        System.out.println("var      = "+var);
                        System.out.println("this.var = "+this.var);
                }
        }

        ClasseInterne ci = this. new ClasseInterne();

        public static void main(String[] args) {
                ClassePrincipale16 cp = new ClassePrincipale16();
                ClasseInterne ci = cp. new ClasseInterne();
                ci.affiche();
        }
}
```

Résultat :

```
C:\>java ClassePrincipale16
var      = 3
this.var = 3
```

Une classe interne a accès à tous les membres de sa classe principale. Dans le code, pour pouvoir faire référence à un membre de la classe principale, il suffit simplement d'utiliser son nom de variable.

Exemple :

```
public class ClassePrincipale17 {
        int valeur = 5;

        class ClasseInterne {
                int var = 3;

                public void affiche() {
                        System.out.println("var      = "+var);
                        System.out.println("this.var = "+this.var);
                        System.out.println("valeur   = "+valeur);
                }
        }

        ClasseInterne ci = this. new ClasseInterne();

        public static void main(String[] args) {
                ClassePrincipale17 cp = new ClassePrincipale17();
                ClasseInterne ci = cp. new ClasseInterne();
                ci.affiche();
        }
}
```

Résultat :

```
C:\testinterne>java ClassePrincipale17
var      = 3
this.var = 3
valeur   = 5
```

La situation se complique un peu plus, si la classe principale et la classe interne possède tous les deux un membre de même nom. Dans ce cas, il faut utiliser la version qualifiée du mot clé this pour accéder au membre de la classe principale. La qualification se fait avec le nom de la classe principale ou plus généralement avec le nom qualifié d'une des classes englobantes.

Exemple :

```
public class ClassePrincipale18 {
        int var = 5;

        class ClasseInterne {
                int var = 3;

                public void affiche() {
                        System.out.println("var                          = "+var);
                        System.out.println("this.var                     = "+this.var);
                        System.out.println("ClassePrincipale18.this.var  = "
                          +ClassePrincipale18.this.var);
                }
        }

        ClasseInterne ci = this. new ClasseInterne();

        public static void main(String[] args) {
                ClassePrincipale18 cp = new ClassePrincipale18();
                ClasseInterne ci = cp. new ClasseInterne();
                ci.affiche();
        }
}
```

Résultat :

```
C:\>java ClassePrincipale18
var                          = 3
this.var                     = 3
ClassePrincipale18.this.var  = 5
```

Comme une classe interne ne peut être nommée du même nom que l'une de ces classes englobantes, ce nom qualifié est unique et il ne risque pas d'y avoir de confusion.

Le nom qualifié d'une classe interne est nom_classe_principale.nom_classe_interne. C'est donc le même principe que celui utilisé pour qualifier une classe contenue dans un package. La notation avec le point est donc légèrement étendue.

L'accès au membre de la classe principale est possible car le compilateur modifie le code de la classe principale et celui de la classe interne pour fournir à la classe interne une référence sur la classe principale.

Le code de la classe interne est modifié pour :

- ajouter une variable privée finale du type de la classe principale nommée this$0
- ajouter un paramètre supplémentaire dans le constructeur qui sera la classe principale et qui va initialiser la variable this$0
- utiliser cette variable pour préfixer les attributs de la classe principale utilisés dans la classe interne.

La code de la classe principale est modifié pour :

- ajouter une méthode static pour chaque champ de la classe principale qui attend en paramètre un objet de la classe principale. Cette méthode renvoie simplement la valeur du champ. Le nom de cette méthode est de la forme access$0
- modifier le code d'instanciation de la classe interne pour appeler le constructeur modifié

Dans le byte code généré, une variable privée finale contient une référence vers la classe principale. Cette variable est nommée this$0. Comme elle est générée par le compilateur, cette variable n'est pas utilisable dans le code source. C'est à partir de cette référence que le compilateur peut modifier le code pour accéder aux membres de la classe principale.

Pour pouvoir avoir accès aux membres de la classe principale, le compilateur génère dans la classe principale des accesseurs sur ces membres. Ainsi, dans la classe interne, pour accéder à un membre de la classe principale, le compilateur appelle un de ces accesseurs en utilisant la référence stockée. Ces méthodes ont un nom de la forme access$numero_unique et sont bien sûr inutilisables dans le code source puisqu'elles sont générées par le compilateur.

En tant que membre de la classe principale, une classe interne peut être déclarée avec le modificateur private ou protected.

Une classe peut faire référence dans le code source à son unique instance lors de l'exécution via le mot clé this. Une classe interne possède au moins deux références :

- l'instance de la classe interne elle même
- éventuellement les instances des classes internes dans laquelle la classe interne est imbriquée
- l'instance de sa classe principale

Dans la classe interne, il est possible pour accéder à une de ces instances d'utiliser le mot clé this préfixé par le nom de la classe suivi d'un point :
nom_classe_principale.this
nom_classe_interne.this

Le mot this seul désigne toujours l'instance de la classe courante dans son code source, donc this seul dans une classe interne désigne l'instance de cette classe interne.

Une classe interne non statique doit toujours être instanciée relativement à un objet implicite ou explicite du type de la classe principale. A la compilation, le compilateur ajoute dans la classe interne une référence vers la classe principale contenu dans une variable privée nommée this$0. Cette référence est initialisée avec un paramètre fourni au constructeur de la classe interne. Ce mécanisme permet de lier les deux instances.

La création d'une classe interne nécessite donc obligatoirement une instance de sa classe principale. Si cette instance n'est pas accessible, il faut en créer une et utiliser une notation particulière de l'opérateur new pour pouvoir instancier la classe interne. Par défaut, lors de l'instanciation d'une classe interne, si aucune instance de la classe principale n'est utilisée, c'est l'instance courante qui est utilisée (mot clé this).

Exemple :

```
public class ClassePrincipale14 {
        class ClasseInterne {
        }

        ClasseInterne ci = this. new ClasseInterne();
}
```

Pour créer une instance d'une classe interne dans une méthode statique de la classe principale, (la méthode main() par exemple), il faut obligatoirement instancier un objet de la classe principale avant et utiliser cet objet lors de la création de l'instance de la classe interne. Pour créer l'instance de la classe interne, il faut alors utiliser une syntaxe particulière de l'opérateur new.

Exemple :

```
public class ClassePrincipale15 {
        class ClasseInterne {
        }

        ClasseInterne ci = this. new ClasseInterne();

        static void maMethode() {
                ClassePrincipale15 cp = new ClassePrincipale15();
                ClasseInterne ci = cp. new ClasseInterne();
        }
}
```

Il est possible d'utiliser une syntaxe condensée pour créer les deux instances en une seule et même ligne de code.

Exemple :

```
public class ClassePrincipale19 {
        class ClasseInterne {
        }

        static void maMethode() {
                ClasseInterne ci = new ClassePrincipale19(). new ClasseInterne();
        }
}
```

Une classe peut hériter d'une classe interne. Dans ce cas, il faut obligatoirement fournir aux constructeurs de la classe une référence sur la classe principale de la classe mère et appeler explicitement dans le constructeur le constructeur de cette classe principale avec une notation particulière du mot clé super

Exemple :

```
public class ClassePrincipale9 {
        public class ClasseInterne {
        }

        class ClasseFille extends ClassePrincipale9.ClasseInterne {
                ClasseFille(ClassePrincipale9 cp) {
                        cp. super();
                }
        }
}
```

Une classe interne peut être déclarée avec les modificateurs final et abstract. Avec le modificateur final, la classe interne ne pourra être utilisée comme classe mère. Avec le modificateur abstract, la classe interne devra être étendue pour pouvoir être instanciée.

4.8.2. Les classes internes locales

Ces classes internes locales (local inner-classes) sont définies à l'intérieure d'une méthode ou d'un bloc de code. Ces classes ne sont utilisables que dans le bloc de code où elles sont définies. Les classes internes locales ont toujours accès aux membres de la classe englobante.

Exemple :

```
public class ClassePrincipale21 {
   int varInstance = 1;

   public static void main(String args[]) {
     ClassePrincipale21 cp = new ClassePrincipale21();
     cp.maMethode();
   }

   public void maMethode() {

     class ClasseInterne {
       public void affiche() {
         System.out.println("varInstance = " + varInstance);
       }
     }

     ClasseInterne ci = new ClasseInterne();
     ci.affiche();
   }
 }
```

Résultat :

```
C:\testinterne>javac ClassePrincipale21.java

C:\testinterne>java ClassePrincipale21
varInstance = 1
```

Leur particularité, en plus d'avoir un accès aux membres de la classe principale, est d'avoir aussi un accès à certaines variables locales du bloc ou est définie la classe interne.

Ces variables définies dans la méthode (variables ou paramètres de la méthode) sont celles qui le sont avec le mot clé final. Ces variables doivent être initialisées avant leur utilisation par la classe interne. Celles ci sont utilisables n'importe où dans le code de la classe interne.

Le modificateur final désigne une variable dont la valeur ne peut être changée une fois qu'elle a été initialisée.

Exemple :

```
public class ClassePrincipale12 {

   public static void main(String args[]) {
     ClassePrincipale12 cp = new ClassePrincipale12();
     cp.maMethode();
   }

   public void maMethode() {
        int varLocale = 3;

     class ClasseInterne {
       public void affiche() {
         System.out.println("varLocale = " + varLocale);
       }
     }

     ClasseInterne ci = new ClasseInterne();
     ci.affiche();
   }
 }
```

Résultat :

```
javac ClassePrincipale12.java
ClassePrincipale12.java:14: Attempt to use a non-final variable varLocale from a
 different method. From enclosing blocks, only final local variables are availab
le.
         System.out.println("varLocale = " + varLocale);
                                             ^
1 error
```

Cette restriction est imposée par la gestion du cycle de vie d'une variable locale. Une telle variable n'existe que durant l'exécution de cette méthode. Une variable finale est une variable dont la valeur ne peut être modifiée après son initialisation. Ainsi, il est possible sans risque pour le compilateur d'ajouter un membre dans la classe interne et de copier le contenu de la variable finale dedans.

Exemple :

```
public class ClassePrincipale13 {

   public static void main(String args[]) {
     ClassePrincipale13 cp = new ClassePrincipale13();
     cp.maMethode();
   }

   public void maMethode() {
        final int varLocale = 3;

     class ClasseInterne {
       public void affiche(final int varParam) {
         System.out.println("varLocale = " + varLocale);
         System.out.println("varParam  = " + varParam);
       }
     }

     ClasseInterne ci = new ClasseInterne();
     ci.affiche(5);
   }
 }
```

Résultat :

```
C:\>javac ClassePrincipale13.java

C:\>java ClassePrincipale13
varLocale = 3
varParam  = 5
```

Pour permettre à une classe interne locale d'accéder à une variable locale utilisée dans le bloc de code ou est définie la classe interne, la variable doit être stockée dans un endroit ou la classe interne pourra y accéder. Pour que cela fonctionne, le compilateur ajoute les variables nécessaires dans le constructeur de la classe interne.

Les variables accédées sont dupliquées dans la classe interne par le compilateur. Il ajoute pour chaque variable un membre privé dans la classe interne dont le nom est de la forme val$nom_variable. Comme la variable accédée est déclarée finale, cette copie peut être faite sans risque. La valeur de chacune de ces variables est fournie en paramètre du constructeur qui a été modifié par le compilateur.

Une classe qui est définie dans un bloc de code n'est pas un membre de la classe englobante : elle n'est donc pas accessible en dehors du bloc de code ou elle est définie. Ces restrictions sont équivalentes à la déclaration d'une variable dans un bloc de code.

Les variables ajoutées par le compilateur sont préfixées par this$ et val$. Ces variables et le constructeur modifié par le compilateur ne sont pas utilisables dans le code source.

Etant visible uniquement dans le bloc de code qui la définie, une classe interne locale ne peut pas utiliser les modificateurs public, private, protected et static dans sa définition. Leur utilisation provoque une erreur à la compilation.

Exemple :

```
public class ClassePrincipale11 {
        public void maMethode() {
                public class ClasseInterne {
                }
        }
}
```

Résultat :

```
javac ClassePrincipale11.java
ClassePrincipale11.java:2: '}' expected.
        public void maMethode() {
                                 ^
ClassePrincipale11.java:3: Statement expected.
                public class ClasseInterne {
                ^
ClassePrincipale11.java:7: Class or interface declaration expected.
}
^
3 errors
```

4.8.3. Les classes internes anonymes

Les classes internes anonymes (anonymous inner-classes) sont des classes internes qui ne possèdent pas de nom. Elles ne peuvent donc être instanciées qu'à l'endroit ou elles sont définies.

Ce type de classe est très pratique lorsqu'une classe doit être utilisée une seule fois : c'est par exemple le cas d'une classe qui doit être utilisée comme un callback.

Une syntaxe particulière de l'opérateur new permet de déclarer et instancier une classe interne :
new classe_ou_interface () {
// définition des attributs et des méthodes de la classe interne
}

Cette syntaxe particulière utilise le mot clé new suivi d'un nom de classe ou interface que la classe interne va respectivement étendre ou implémenter. La définition de la classe suit entre deux accolades. Une classe interne anonyme peut soit hériter d'une classe soit implémenter une interface mais elle ne peut pas explicitement faire les deux.

Si la classe interne étend une classe, il est possible de fournir des paramètres entre les parenthèses qui suivent le nom de la classe. Ces arguments éventuels fournis au moment de l'utilisation de l'opérateur new sont passés au constructeur de la super classe. En effet, comme la classe ne possède pas de nom, elle ne possède pas non plus de constructeur.

Les classes internes anonymes qui implémentent une interface héritent obligatoirement de classe Object. Comme cette classe ne possède qu'un constructeur sans paramètre, il n'est pas possible lors de l'instanciation de la classe interne de lui fournir des paramètres.

Une classe interne anonyme ne peut pas avoir de constructeur puisqu'elle ne possède pas de nom mais elle peut avoir des initialisateurs.

Exemple :

```
 public void init() {
   boutonQuitter.addActionListener(
     new ActionListener() {
       public void actionPerformed(ActionEvent e) {
         System.exit(0);
       }
```

```
      }
    );
  }
```

Les classes anonymes sont un moyen pratique de déclarer un objet sans avoir à lui trouver un nom. La contre partie est que cette classe ne pourra être instanciée dans le code qu'à l'endroit ou elle est définie : elle est déclarée et instanciée en un seul et unique endroit.

Le compilateur génère un fichier ayant pour nom la forme suivante : nom_classe_principale$numéro_unique. En fait, le compilateur attribut un numéro unique à chaque classe interne anonyme et c'est ce numéro qui est donné au nom du fichier préfixé par le nom de la classe englobante et d'un signe '$'.

4.8.4. Les classes internes statiques

Les classes internes statiques (static member inner-classes) sont des classes internes qui ne possèdent pas de référence vers leur classe principale. Elles ne peuvent donc pas accéder aux membres d'instance de leur classe englobante. Elles peuvent toutefois avoir accès aux variables statiques de la classe englobante.

Pour les déclarer, il suffit d'utiliser en plus le modificateur static dans la déclaration de la classe interne.

Leur utilisation est obligatoire si la classe est utilisée dans une méthode statique qui par définition peut être appelée dans avoir d'instance de la classe et que l'on ne peut pas avoir une instance de la classe englobante. Dans le cas contraire, le compilateur indiquera une erreur :

Exemple :

```
public class ClassePrincipale4 {

        class ClasseInterne {
                public void afficher() {
                        System.out.println("bonjour");
                }
        }

        public static void main(String[] args) {
                new ClasseInterne().afficher();
        }
}
```

Résultat :

```
javac ClassePrincipale4.java
ClassePrincipale4.java:10: No enclosing instance of class ClassePrincipale4 is i
n scope; an explicit one must be provided when creating inner class ClassePrinci
pale4. ClasseInterne, as in "outer. new Inner()" or "outer. super()".
                new ClasseInterne().afficher();
                ^
1 error
```

En déclarant la classe interne static, le code se compile et peut être exécuté

Exemple :

```
public class ClassePrincipale4 {

        static class ClasseInterne {
                public void afficher() {
                        System.out.println("bonjour");
                }
        }

        public static void main(String[] args) {
```

```
                new ClasseInterne().afficher();
        }
}
```

Résultat :

```
javac ClassePrincipale4.java

java ClassePrincipale4
bonjour
```

Comme elle ne possède pas de référence sur sa classe englobante, une classe interne statique est en fait traduite par le compilateur comme une classe principale. En fait, il est difficile de les mettre dans une catégorie (classe principale ou classe interne) car dans le code source c'est une classe interne (classe définie dans une autre) et dans le byte code généré c'est une classe principale.

Ce type de classe n'est pas très employé.

4.9. La gestion dynamique des objets

Tout objet appartient à une classe et Java sait la reconnaitre dynamiquement.

Java fournit dans son API un ensemble de classes qui permettent d'agir dynamiquement sur des classes. Cette technique est appelée introspection et permet :

- de décrire une classe ou une interface : obtenir son nom, sa classe mère, la liste de ses méthodes, de ses variables de classe, de ses constructeurs et de ses variables d'instances
- d'agir sur une classe en envoyant, à un objet Class des messages comme à tout autre objet. Par exemple, créer dynamiquement à partir d'un objet Class une nouvelle instance de la classe représentée

Voir le chapitre «La gestion dynamique des objets et l'introspection» pour obtenir plus d'informations.

5. Les packages de bases

Le JDK se compose de nombreuses classes regroupées selon leur fonctionnalité en packages. La première version du JDK était composée de 8 packages qui constituent encore aujourd'hui les packages de bases des différentes versions du JDK.

Ce chapitre contient plusieurs sections :

- Les packages selon la version du JDK
- Le package java.lang
- La présentation rapide du package awt java
- La présentation rapide du package java.io
- Le package java.util
- La présentation rapide du package java.net
- La présentation rapide du package java.applet

5.1. Les packages selon la version du JDK

Selon sa version, le JDK contient un certain nombre de packages, chacun étant constitué par un ensemble de classes qui couvrent un même domaine et apportent de nombreuses fonctionnalités. Les différentes versions du JDK sont constamment enrichis avec de nouveaux packages :

Packages	JDK 1.0	JDK 1.1	JDK 1.2	JDK 1.3	JDK 1.4	JDK 1.5	JDK 6.0
java.applet Développement des applets	X	X	X	X	X	X	X
java.awt Toolkit pour interfaces graphiques	X	X	X	X	X	X	X
java.awt.color Gérer et utiliser les couleurs			X	X	X	X	X
java.awt.datatransfer Echanger des données via le presse-papier		X	X	X	X	X	X
java.awt.dnd Gérer le cliquer/glisser			X	X	X	X	X
java.awt.event Gérer les événements utilisateurs		X	X	X	X	X	X
java.awt.font Utiliser les fontes			X	X	X	X	X
java.awt.geom dessiner des formes géométriques			X	X	X	X	X
java.awt.im				X	X	X	X

java.awt.im.spi				X	X	X	X
java.awt.image Afficher des images		X	X	X	X	X	X
java.awt.image.renderable Modifier le rendu des images			X	X	X	X	X
java.awt.print Réaliser des impressions			X	X	X	X	X
java.beans Développer des composants réutilisables		X	X	X	X	X	X
java.beans.beancontext			X	X	X	X	X
java.io Gérer les flux	X	X	X	X	X	X	X
java.lang Classes de base du langage	X	X	X	X	X	X	X
java.lang.annotation						X	X
java.lang.instrument						X	X
java.lang.management						X	X
java.lang.ref			X	X	X	X	X
java.lang.reflect Utiliser la réflexion (introspection)		X	X	X	X	X	X
java.math Utiliser des opérations mathématiques		X	X	X	X	X	X
java.net Utiliser les fonctionnalités réseaux	X	X	X	X	X	X	X
java.nio					X	X	X
java.nio.channels					X	X	X
java.nio.channels.spi					X	X	X
java.nio.charset					X	X	X
java.nio.charset.spi					X	X	X
java.rmi Développement d'objets distribués		X	X	X	X	X	X
java.rmi.activation			X	X	X	X	X
java.rmi.dgc		X	X	X	X	X	X
java.rmi.registry		X	X	X	X	X	X
java.rmi.server Gérer les objets serveurs de RMI		X	X	X	X	X	X
java.security Gérer les signatures et les certifications		X	X	X	X	X	X
java.security.acl		X	X	X	X	X	X
java.security.cert		X	X	X	X	X	X
java.security.interfaces		X	X	X	X	X	X
java.security.spec			X	X	X	X	X
		X	X	X	X	X	X

java.sql JDBC pour l'accès aux bases de données							
java.text Formater des objets en texte		X	X	X	X	X	X
java.text.spi							X
java.util Utilitaires divers	X	X	X	X	X	X	X
java.util.concurrent						X	X
java.util.concurrent.atomic						X	X
java.util.cocurrent.locks						X	X
java.util.jar Gérer les fichiers jar			X	X	X	X	X
java.util.logging Utiliser des logs					X	X	X
java.util.prefs Gérer des préférences					X	X	X
java.util.regex Utiliser les expressions régulières					X	X	X
java.util.spi							X
java.util.zip Gérer les fichiers zip		X	X	X	X	X	X
javax.accessibility			X	X	X	X	X
javax.activation							X
javax.activity						X	X
javax.annotation							X
javax.annotation.processing							X
javax.crypto Utiliser le cryptage des données					X	X	X
javax.crypto.interfaces					X	X	X
javax.crypto.spec					X	X	X
javax.imageio					X	X	X
javax.imageio.event					X	X	X
javax.imageio.metadata					X	X	X
javax.imageio.plugins.bmp						X	X
javax.imageio.plugins.jpeg					X	X	X
javax.imageio.spi					X	X	X
javax.imageio.stream					X	X	X
javax.jws							X
javax.jws.soap							X
javax.lang.model							X
javax.lang.model.element							X
javax.lang.model.type							X

javax.lang.model.util							X
javax.management						X	X
javax.management.loading						X	X
javax.management.modelmbean						X	X
javax.management.monitor						X	X
javax.management.openmbean						X	X
javax.management.relation						X	X
javax.management.remote						X	X
javax.management.remote.rmi						X	X
javax.management.timer						X	X
javax.naming				X	X	X	X
javax.naming.directory				X	X	X	X
javax.naming.event				X	X	X	X
javax.naming.ldap				X	X	X	X
javax.naming.spi				X	X	X	X
javax.net					X	X	X
javax.net.ssl Utiliser une connexion réseau sécurisée avec SSL					X	X	X
javax.print					X	X	X
javax.print.attribute					X	X	X
javax.print.attribute.standard					X	X	X
javax.print.event					X	X	X
javax.rmi				X	X	X	X
javax.rmi.CORBA				X	X	X	X
javax.rmi.ssl						X	X
javax.script							X
javax.security.auth API JAAS pour l'authentification et l'autorisation					X	X	X
javax.security.auth.callback					X	X	X
javax.security.auth.kerberos					X	X	X
javax.security.auth.login					X	X	X
javax.security.auth.spi					X	X	X
javax.security.auth.x500					X	X	X
javax.security.cert					X	X	X
javax.security.sasl						X	X
javax.sound.midi				X	X	X	X
javax.sound.midi.spi				X	X	X	X
javax.sound.sampled				X	X	X	X
javax.sound.sampled.spi				X	X	X	X

javax.sql					X	X	X
javax.sql.rowset						X	X
javax.sql.rowset.serial						X	X
javax.sql.rowset.spi						X	X
javax.swing Swing pour développer des interfaces graphiques			X	X	X	X	X
javax.swing.border Gérer les bordures des composants Swing			X	X	X	X	X
javax.swing.colorchooser Composant pour sélectionner une couleur			X	X	X	X	X
javax.swing.event Gérer des événements utilisateur des composants Swing			X	X	X	X	X
javax.swing.filechooser Composant pour sélectionner un fichier			X	X	X	X	X
javax.swing.plaf Gérer l'aspect des composants Swing			X	X	X	X	X
javax.swing.plaf.basic			X	X	X	X	X
javax.swing.plaf.metal Gérer l'aspect metal des composants Swing			X	X	X	X	X
javax.swing.plaf.multi			X	X	X	X	X
javax.swing.plaf.synth						X	X
javax.swing.table			X	X	X	X	X
javax.swing.text			X	X	X	X	X
javax.swing.text.html			X	X	X	X	X
javax.swing.text.html.parser			X	X	X	X	X
javax.swing.text.rtf			X	X	X	X	X
javax.swing.tree Un composant de type arbre			X	X	X	X	X
javax.swing.undo Gérer les annulations d'opérations d'édition			X	X	X	X	X
javax.tools							X
javax.transaction				X	X	X	X
javax.transaction.xa					X	X	X
javax.xml						X	X
javax.xml.bind							X
javax.xml.bind.annotation							X
javax.xml.bind.annotation.adapters							X
javax.xml.bind.attachment							X
javax.xml.bind.helpers							X
javax.xml.bind.util							X
javax.xml.crypto							X

javax.xml.crypto.dom							X
javax.xml.crypto.dsig							X
javax.xml.crypto.dsig.dom							X
javax.xml.crypto.dsig.keyinfo							X
javax.xml.crypto.dsig.spec							X
javax.xml.datatype						X	X
javax.xml.namespace						X	X
javax.xml.parsers API JAXP pour utiliser XML					X	X	X
javax.xml.soap							X
javax.xml.stream							X
javax.xml.stream.events							X
javax.xml.stream.events							X
javax.xml.transform transformer un document XML avec XSLT					X	X	X
javax.xml.transform.dom					X	X	X
javax.xml.transform.sax					X	X	X
javax.xml.transform.stream					X	X	X
javax.xml.validation						X	X
javax.xml.ws							X
javax.xml.ws.handler							X
javax.xml.ws.handler.soap							X
javax.xml.ws.http							X
javax.xml.ws.soap							X
javax.xml.ws.spi							X
javax.xml.xpath						X	X
org.ietf.jgss					X	X	X
org.omg.CORBA			X	X	X	X	X
org.omg.CORBA_2_3				X	X	X	X
org.omg.CORBA_2_3.portable				X	X	X	X
org.omg.CORBA.DynAnyPackage			X	X	X	X	X
org.omg.CORBA.ORBPackage			X	X	X	X	X
org.omg.CORBA.portable			X	X	X	X	X
org.omg.CORBA.TypeCodePackage			X	X	X	X	X
org.omg.CosNaming			X	X	X	X	X
org.omg.CosNaming.NamingContextExtPackage			X	X	X	X	X
org.omg.CosNaming.NamingContextPackage					X	X	X
org.omg.Dynamic					X	X	X
org.omg.DynamicAny					X	X	X

org.omg.DynamicAny.DynAnyFactoryPackage					X	X	X
org.omg.DynamicAny.DynAnyPackage					X	X	X
org.omg.IOP					X	X	X
org.omg.IOP.CodecFactoryPackage					X	X	X
org.omg.IOP.CodecPackage					X	X	X
org.omg.Messaging					X	X	X
org.omg.PortableInterceptor					X	X	X
org.omg.PortableInterceptor.ORBInitInfoPackage					X	X	X
org.omg.PortableServer					X	X	X
org.omg.PortableServer.CurrentPackage					X	X	X
org.omg.PortableServer.POAPackage					X	X	X
org.omg.PortableServer.ServantLocatorPackage					X	X	X
org.omg.PortableServer.portable					X	X	X
org.omg.SendingContext				X	X	X	X
org.omg.stub.java.rmi				X	X	X	X
org.w3c.dom Utiliser DOM pour un document XML					X	X	X
org.w3c.dom.bootstrap						X	X
org.w3c.dom.events						X	X
org.w3c.dom.ls						X	X
org.xml.sax Utiliser SAX pour un document XML					X	X	X
org.xml.sax.ext					X	X	X
org.xml.sax.helpers					X	X	X

5.2. Le package java.lang

Ce package de base contient les classes fondamentales tel que Object, Class, Math, System, String, StringBuffer, Thread, les wrappers etc ... Certaines de ces classes sont détaillées dans les sections suivantes.

Il contient également plusieurs classes qui permettent de demander des actions au système d'exploitation sur laquelle la machine virtuelle tourne, par exemple les classes ClassLoader, Runtime, SecurityManager.

Certaines classes sont détaillées dans des chapitres dédiés : la classe Math est détaillée dans le chapitre «Les fonctions mathématiques», la classe Class est détaillée dans le chapitre «La gestion dynamique des objets et l'introspection» et la classe Thread est détaillée dans le chapitre «Le multitâche».

Ce package est tellement fondamental qu'il est implicitement importé dans tous les fichiers sources par le compilateur.

5.2.1. La classe Object

C'est la super classe de toutes les classes Java : toutes ces méthodes sont donc héritées par toutes les classes.

5.2.1.1. La méthode getClass()

La méthode getClass() renvoie un objet de la classe Class qui représente la classe de l'objet.

Le code suivant permet de connaitre le nom de la classe de l'objet

Exemple :

```
String nomClasse = monObject.getClass().getName();
```

5.2.1.2. La méthode toString()

La méthode toString() de la classe Object renvoie le nom de la classe , suivi du séparateur @, lui même suivi par la valeur de hachage de l'objet.

5.2.1.3. La méthode equals()

La méthode equals() implémente une comparaison par défaut. Sa définition dans Object compare les références : donc obj1.equals(obj2) ne renverra true que si obj1 et obj2 désignent le même objet. Dans une sous classe de Object, pour laquelle on a besoin de pouvoir dire que deux objets distincts peuvent être égaux, il faut redéfinir la méthode equals() héritée de Object.

5.2.1.4. La méthode finalize()

A l'inverse de nombreux langages orientés objet tel que le C++ ou Delphi, le programmeur Java n'a pas à se préoccuper de la destruction des objets qu'il instancie. Ceux-ci sont détruits et leur emplacement mémoire est récupéré par le ramasse miette de la machine virtuelle dès qu'il n'y a plus de référence sur l'objet.

La machine virtuelle garantit que toutes les ressources Java sont correctement libérées mais, quand un objet encapsule une ressource indépendante de Java (comme un fichier par exemple), il peut être préférable de s'assurer que la ressource sera libérée quand l'objet sera détruit. Pour cela, la classe Object définit la méthode protected finalize(), qui est appelée quand le ramasse miettes doit récupérer l'emplacement de l'objet ou quand la machine virtuelle termine son exécution

Exemple :

```
import java.io.*;

public class AccesFichier {
   private FileWriter fichier;

   public AccesFichier(String s) {
      try {
      fichier = new FileWriter(s);
      }
      catch (IOException e) {
         System.out.println("Impossible d'ouvrir le fichier");
      }
   }

   protected void finalize() throws Throwable {
      super.finalize(); // obligatoire : appel finalize heritee
      System.out.println("Appel de la méthode finalize");
      termine();
   }

   public static void main(String[] args) {
      AccesFichier af = new AccesFichier("c:\test");
      System.exit(0);
   }
```

```
   public void termine() {
      if (fichier != null) {
         try {
            fichier.close();
         }
         catch (IOException e) {
            System.out.println("Impossible de fermer le fichier");
         }
         fichier = null;
      }
   }
}
```

5.2.1.5. La méthode clone()

Si x désigne un objet obj1, l'exécution de x.clone() renvoie un second objet obj2, qui est une copie de obj1 : si obj1 est ensuite modifié, obj2 n'est pas affecté par ce changement.

Par défaut, la méthode clone(), héritée de Object fait une copie variable par variable : elle offre donc un comportement acceptable pour de très nombreuses sous classe de Object. Cependant comme le processus de duplication peut être délicat à gérer pour certaines classes (par exemple des objets de la classe Container), l'héritage de clone ne suffit pas pour qu'une classe supporte le clonage.

Pour permettre le clonage d'une classe, il faut implémenter dans la classe l'interface Cloneable.

La première chose que fait la méthode clone() de la classe Object, quand elle est appelée, est de tester si la classe implémente Cloneable. Si ce n'est pas le cas, elle lève l'exception CloneNotSupportedException.

5.2.2. La classe String

Une chaîne de caractères est contenue dans un objet de la classe String

On peut initialiser une variable String sans appeler explicitement un constructeur : le compilateur se charge de créer un objet.

Exemple : deux déclarations de chaines identiques.

```
String uneChaine = "bonjour";
String uneChaine = new String("bonjour");
```

Les objets de cette classe ont la particularité d'être constants. Chaque traitement qui vise à transformer un objet de la classe est implémenté par une méthode qui laisse l'objet d'origine inchangé et renvoie un nouvel objet String contenant les modifications.

Exemple :

```
private String uneChaine;
void miseEnMajuscule(String chaine) {
    uneChaine = chaine.toUpperCase()
```

Il est ainsi possible d'enchainer plusieurs méthodes :

Exemple :

```
uneChaine = chaine.toUpperCase().trim();
```

L'opérateur + permet la concaténation de chaines de caractères.

La comparaison de deux chaînes doit se faire via la méthode equals() qui compare les objets eux même et non l'opérateur == qui compare les références de ces objets :

Exemple :

```
String nom1 = new String("Bonjour");
String nom2 = new String("Bonjour");
System.out.println(nom1 == nom2); // affiche false
System.out.println( nom1.equals(nom2)); // affiche true
```

Cependant dans un souci d'efficacité, le compilateur ne duplique pas 2 constantes chaines de caractères : il optimise l'espace mémoire utilisé en utilisant le même objet. Cependant, l'appel explicite du constructeur ordonne au compilateur de créer un nouvel objet.

Exemple :

```
String nom1 = "Bonjour";
String nom2 = "Bonjour";
String nom3 = new String("Bonjour");
System.out.println(nom1 == nom2); // affiche true
System.out.println(nom1 == nom3); // affiche false
```

La classe String possède de nombreuses méthodes dont voici les principales :

Méthodes la classe String	Rôle
charAt(int)	renvoie le nième caractère de la chaine
compareTo(String)	compare la chaîne avec l'argument
concat(String)	ajoute l'argument à la chaîne et renvoie la nouvelle chaîne
endsWith(String)	vérifie si la chaîne se termine par l'argument
equalsIgnoreCase(String)	compare la chaîne sans tenir compte de la casse
indexOf(String)	renvoie la position de début à laquelle l'argument est contenu dans la chaine
lastIndexOf(String)	renvoie la dernière position à laquelle l'argument est contenu dans la chaine
lenght()	renvoie la longueur de la chaine
replace(char,char)	renvoie la chaîne dont les occurrences d'un caractère sont remplacées
startsWith(String int)	Vérifie si la chaîne commence par la sous chaîne
substring(int,int)	renvoie une partie de la chaine
toLowCase()	renvoie la chaîne en minuscule
toUpperCase()	renvoie la chaîne en majuscule
trim()	enlève les caractères non significatifs de la chaine

La méthode isEmtpy() ajoutée dans Java SE 6 facilite le test d'une chaîne de caractères vide.

Cette méthode utilise les données de l'instance de l'objet, il est donc nécessaire de vérifier que cette instance n'est pas null pour éviter la levée d'une exception de type NullPointerException.

Exemple :

```
package com.jmdoudoux.test.java6;
```

```
public class TestEmptyString {

    public static void main(String args[]) {

        String chaine = null;
        try {
            if (chaine.isEmpty()){
                System.out.println("la chaine est vide");
            }
        } catch (Exception e) {
            System.out.println("la chaine est null");
        }

        chaine = "test";
        if (chaine.isEmpty()){
            System.out.println("la chaine est vide");
        } else  {
            System.out.println("la chaine n'est pas vide");
        }

        chaine = "";
        if (chaine.isEmpty()){
            System.out.println("la chaine est vide");
        } else  {
            System.out.println("la chaine n'est pas vide");
        }
    }
}
```

Résultat :

```
la chaine est null
la chaine n'est pas vide
la chaine est vide
```

5.2.3. La classe StringBuffer

Les objets de cette classe contiennent des chaines de caractères variables, ce qui permet de les agrandir ou de les réduire. Cet objet peut être utilisé pour construire ou modifier une chaîne de caractères chaque fois que l'utilisation de la classe String nécessiterait de nombreuses instanciations d'objets temporaires.

Par exemple, si str est un objet de type String, le compilateur utilisera la classe StringBuffer pour traiter la concaténation de "abcde"+str+"z" en générant le code suivant : new StringBuffer().append("abcde").append(str).append("z").toString();

Ce traitement aurait pu être réalisé avec trois appels à la méthode concat() de la classe String mais chacun des appels aurait instancié un objet StringBuffer pour réaliser la concaténation, ce qui est couteux en temps exécution

La classe StringBuffer dispose de nombreuses méthodes qui permettent de modifier le contenu de la chaîne de caractère

Exemple (code Java 1.1) :

```
public class MettreMaj {

   static final String lMaj = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
   static final String lMin = "abcdefghijklmnopqrstuvwxyz";

   public static void main(java.lang.String[] args) {
      System.out.println(MetMaj("chaine avec MAJ et des min"));
   }

   public static String MetMaj(String s) {
      StringBuffer sb = new StringBuffer(s);

      for ( int i = 0; i < sb.length(); i++) {
          int index = lMin.indexOf(sb.charAt(i));
```

```
            if (index <=0 ) sb.setCharAt(i,lMaj.charAt(index));
        }
        return sb.toString();
    }
}
```

Résultat :

```
CHAINE AVEC MAJ ET DES MIN
```

5.2.4. Les wrappers

Les objet de type wrappers (enveloppeurs) représentent des objets qui encapsulent une donnée de type primitif et qui fournissent un ensemble de méthodes qui permettent notamment de faire des conversions.

Ces classes offrent toutes les services suivants :

- un constructeur qui permet une instanciation à partir du type primitif et un constructeur qui permet une instanciation à partir d'un objet String
- une méthode pour fournir la valeur primitive représentée par l'objet
- une méthode equals() pour la comparaison.

Les méthodes de conversion opèrent sur des instances, mais il est possible d'utiliser des méthodes statiques.

Exemple :

```
int valeur =
Integer.valueOf("999").intValue();
```

Ces classes ne sont pas interchangeables avec les types primitifs d'origine car il s'agit d'objet.

Exemple :

```
Float objetpi = new Float("3.1415");
System.out.println(5*objetpi); // erreur à la compil
```

Pour obtenir la valeur contenue dans l'objet, il faut utiliser la méthode typeValue() ou type est le nom du type standard

Exemple :

```
Integer Entier = new Integer("10");
int entier = Entier.intValue();
```

Les classes Integer, Long, Float et Double définissent toutes les constantes MAX_VALUE et MIN_VALUE qui représentent leurs valeurs minimales et maximales.

Lorsque l'on effectue certaines opérations mathématiques sur des nombres à virgules flottantes (float ou double), le résultat peut prendre l'une des valeurs suivantes :

- NEGATIVE_INFINITY : infini négatif causé par la division d'un nombre négatif par 0.0
- POSITIVE_INFINITY : infini positif causé par la division d'un nombre positif par 0.0
- NaN: n'est pas un nombre (Not a Number) causé par la division de 0.0 par 0.0

Il existe des méthodes pour tester le résultat :

Float.isNaN(float); //idem avec double
Double.isInfinite(double); // idem avec float

Exemple :

```
float res = 5.0f / 0.0f;

if (Float.isInfinite(res)) { ... };
```

La constante Float.NaN n'est ni égale à un nombre dont la valeur est NaN ni à elle même. Float.NaN == Float.NaN retourne False

Lors de la division par zéro d'un nombre entier, une exception est levée.

Exemple :

```
System.out.println(10/0);

Exception in thread "main" java.lang.ArithmeticException: / by zero
at test9.main(test9.java:6)
```

5.2.5. La classe System

Cette classe possède de nombreuses fonctionnalités pour utiliser des services du système d'exploitation.

5.2.5.1. L'utilisation des flux d'entrée/sortie standard

La classe System défini trois variables statiques qui permettent d'utiliser les flux d'entrée/sortie standards du système d'exploitation.

Variable	Type	Rôle
in	InputStream	Entrée standard du système. Par défaut, c'est le clavier.
out	PrintStream	Sortie standard du système. Par défaut, c'est le moniteur.
err	PrintStream	Sortie standard des erreurs du système. Par défaut, c'est le moniteur.

Exemple :

```
System.out.println("bonjour");
```

La classe système possède trois méthodes qui permettent de rediriger ces flux.

La méthode printf() dont le mode fonctionnement bien connu dans le langage C a été reprise pour être ajoutée dans l'API de Java.

Exemple :

```
public class TestPrintf {
  public static void main(String[] args) {
    System.out.printf("%4d",32);
  }
}
```

La méthode printf propose :

- un nombre d'arguments variable
- des formats standards pour les types primitifs, String et Date
- des justifications possibles avec certains formats
- l'utilisation de la localisation pour les données numériques et de type date

Exemple (java 1.5):

```
import java.util.*;

public class TestPrintf2 {

  public static void main(String[] args) {
    System.out.printf("%d \n"                       ,13);
    System.out.printf("%4d \n"                      ,13);
    System.out.printf("%04d \n"                     ,13);
    System.out.printf("%f \n"                       ,3.14116);
    System.out.printf("%.2f \n"                     ,3.14116);
    System.out.printf("%s \n"                       ,"Test");
    System.out.printf("%10s \n"                     ,"Test");
    System.out.printf("%-10s \n"                    ,"Test");
    System.out.printf("%tD \n"                      , new Date());
    System.out.printf("%tF \n"                      , new Date());
    System.out.printf("%1$te %1$tb %1$ty \n"        , new Date());
    System.out.printf("%1$tA %1$te %1$tB %1$tY \n", new Date());
    System.out.printf("%1$tr \n"                    , new Date());
  }

}
```

Résultat :

```
C:\tiger>java TestPrintf2
13
13
0013
3,141160
3,14
Test
Test
Test
08/23/04
2004-08-23
23 ao¹t 04
lundi 23 ao¹t 2004
03:56:25 PM
```

Une exception est levée lors de l'exécution si un des formats utilisés est inconnu.

Exemple (java 1.5):

```
C:\tiger>java TestPrintf2
13 1300133,1411603,14Test TestTest 08/23/04Exception in thread "main"
java.util.UnknownFormatConversionException: Conversion = 'tf'
at java.util.Formatter$FormatSpecifier.checkDateTime(Unknown Source)
at java.util.Formatter$FormatSpecifier.<init>(Unknown Source)
at java.util.Formatter.parse(Unknown Source)
at java.util.Formatter.format(Unknown Source)
at java.io.PrintStream.format(Unknown Source)
at java.io.PrintStream.printf(Unknown Source)
at TestPrintf2.main(TestPrintf2.java:15)
```

5.2.5.2. Les variables d'environnement et les propriétés du système

JDK 1.0 propose la méthode statique getEnv() qui renvoie la valeur de la propriété système dont le nom est fourni en paramètre.

Depuis le JDK 1.1, cette méthode est deprecated car elle n'est pas très portable. Son utilisation lève une exception :

Exemple :

```
java.lang.Error: getenv no longer  supported, use properties and -D instead: windir
      at  java.lang.System.getenv(System.java:691)
      at  com.jmd.test.TestPropertyEnv.main(TestPropertyEnv.java:6)
Exception  in thread "main"
```

Elle est remplacée par un autre mécanisme qui n'interroge pas directement le système mais qui recherche les valeurs dans un ensemble de propriétés. Cet ensemble est constitué de propriétés standards fournies par l'environnement java et par des propriétés ajoutées par l'utilisateur. Jusqu'au JDK 1.4, il est nécessaire d'utiliser ces propriétés de la JVM.

Voici une liste non exhaustive des propriétés fournies par l'environnement java :

Nom de la propriété	Rôle
java.version	Version du JRE
java.vendor	Auteur du JRE
java.vendor.url	URL de l'auteur
java.home	Répertoire d'installation de java
java.vm.version	Version de l'implémentation la JVM
java.vm.vendor	Auteur de l'implémentation de la JVM
java.vm.name	Nom de l'implémentation de la JVM
java.specification.version	Version des spécifications de la JVM
java.specification.vendor	Auteur des spécifications de la JVM
java.specification.name	Nom des spécifications de la JVM
java.ext.dirs	Chemin du ou des répertoires d'extension
os.name	Nom du système d'exploitation
os.arch	Architecture du système d'exploitation
os.version	Version du système d'exploitation
file.separator	Séparateur de fichiers (exemple : "/" sous Unix, "\" sous Windows)
path.separator	Séparateur de chemin (exemple : ":" sous Unix, ";" sous Windows)
line.separator	Séparateur de ligne
user.name	Nom du user courant
user.home	Répertoire d'accueil du user courant
user.dir	Répertoire courant au moment de l'initialisation de la propriété

Exemple :

```
public class TestProperty {

  public static void main(String[] args) {
    System.out.println("java.version                ="+System.getProperty("java.version"));
    System.out.println("java.vendor                 ="+System.getProperty("java.vendor"));
    System.out.println("java.vendor.url             ="+System.getProperty("java.vendor.url"));
```

```
    System.out.println("java.home                    ="+System.getProperty("java.home"));
    System.out.println("java.vm.specification.version ="
      +System.getProperty("java.vm.specification.version"));
    System.out.println("java.vm.specification.vendor  ="
      +System.getProperty("java.vm.specification.vendor"));
    System.out.println("java.vm.specification.name  ="
      +System.getProperty("java.vm.specification.name"));
    System.out.println("java.vm.version             ="+System.getProperty("java.vm.version"));
    System.out.println("java.vm.vendor              ="+System.getProperty("java.vm.vendor"));
    System.out.println("java.vm.name                ="+System.getProperty("java.vm.name"));
    System.out.println("java.specification.version  ="
      +System.getProperty("java.specification.version"));
    System.out.println("java.specification.vendor   ="
      +System.getProperty("java.specification.vendor"));
    System.out.println("java.specification.name     ="
      +System.getProperty("java.specification.name"));
    System.out.println("java.class.version          ="
      +System.getProperty("java.class.version"));
    System.out.println("java.class.path             ="
      +System.getProperty("java.class.path"));
    System.out.println("java.ext.dirs               ="+System.getProperty("java.ext.dirs"));
    System.out.println("os.name                     ="+System.getProperty("os.name"));
    System.out.println("os.arch                     ="+System.getProperty("os.arch"));
    System.out.println("os.version                  ="+System.getProperty("os.version"));
    System.out.println("file.separator              ="+System.getProperty("file.separator"));
    System.out.println("path.separator              ="+System.getProperty("path.separator"));
    System.out.println("line.separator              ="+System.getProperty("line.separator"));
    System.out.println("user.name                   ="+System.getProperty("user.name"));
    System.out.println("user.home                   ="+System.getProperty("user.home"));
    System.out.println("user.dir                    ="+System.getProperty("user.dir"));
  }
}
```

Par défaut, l'accès aux propriétés système est restreint par le SecurityManager pour les applets.

Pour définir ces propres propriétés, il faut utiliser l'option -D de l'interpréteur java en utilisant la ligne de commande.

La méthode statique getProperty() permet d'obtenir la valeur de la propriété dont le nom est fourni en paramètre. Une version surchargée de cette méthode permet de préciser un second paramètre qui contiendra la valeur par défaut, si la propriété n'est pas définie.

Exemple : obtenir une variable système (java 1.1, 1.2, 1.3 et 1.4)

```
package com.jmd.test;

public class TestPropertyEnv {

  public static void main(String[] args) {
    System.out.println("env.windir ="+System.getProperty("env.windir"));
  }
}
```

Exemple : Execution

```
C:\tests>java -Denv.windir=%windir% -cp . com.jmd.test.TestPropertyEnv
env.windir =C:\WINDOWS
```

Java 5 propose de nouveau une implémentation pour la méthode System.getEnv() possédant deux surcharges :

- Une sans paramètre qui renvoie une collection des variables système
- Une avec un paramètre de type string qui contient le nom de la variable à obtenir

Exemple (Java 5):

```
package com.jmd.tests;
```

```
public class TestPropertyEnv {

    public static void main(String[] args) {
      System.out.println(System.getenv("windir"));
    }
}
```

La surcharge sans argument permet d'obtenir une collection de type Map contenant les variables d'environnement système.

Exemple (Java 5) :

```
package com.jmd.tests;

import java.util.Iterator;
import java.util.Map;
import java.util.Set;

public class TestPropertyEnv {

    public static void main(String[] args) {
      Map map = System.getenv();
      Set cles = map.keySet();
      Iterator iterator = cles.iterator();
      while (iterator.hasNext()) {
        String cle = (String) iterator.next();
        System.out.println(cle+" : "+map.get(cle));
      }
    }
}
```

5.2.6. Les classes Runtime et Process

La classe Runtime permet d'interagir avec le système dans lequel l'application s'exécute : obtenir des informations sur le système, arrêt de la machine virtuelle, exéecution d'un programme externe.

Cette classe ne peut pas être instanciée mais il est possible d'obtenir une instance en appelant la méthode statique getRuntime() de la classe RunTime.

Les méthodes totalMemory() et freeMemory() permettent d'obtenir respectivement la quantité totale de la mémoire et la quantité de mémoire libre.

Exemple :

```
package com.jmd.tests;

public class TestRuntime1 {

    public static void main(String[] args) {
        Runtime runtime = Runtime.getRuntime();
        System.out.println("Mémoire totale = " + runtime.totalMemory());
        System.out.println("Memoire libre = " + runtime.freeMemory());
    }
}
```

La méthode exec() permet d'exécuter des processus sur le système d'exploitation ou s'exécute la JVM. Elle lance la commande de manière asynchrone et renvoie un objet de type Process pour obtenir des informations sur le processus lancé.

Il existe plusieurs surcharges de cette méthode pouvant toutes entre autre lever une exception de type SecurityException, IOException, NullPointerException :

Méthode	Remarque
Process exec(String command)	
Process exec(String[] cmdarray)	
Process exec(String[] cmdarray, String[] envp)	
Process exec(String[] cmdarray, String[] envp, File dir)	(depuis Java 1.3)
Process exec(String cmd, String[] envp)	
Process exec(String command, String[] envp, File dir)	(depuis Java 1.3)

La commande à exécuter peut être fournie sous la forme d'une chaîne de caractères ou sous la forme d'un tableau dont le premier élément est la commande et les éléments suivants sont ses arguments. Deux des surcharges accepte un objet de File qui encapsule le répertoire dans lequel la commande va être exécutée.

Important : la commande exec() n'est pas un interpréteur de commande. Il n'est par exemple pas possible de préciser dans la commande une redirection vers un fichier. Ainsi pour exécuter une commande de l'interpréteur DOS sous Windows, il est nécessaire de préciser l'interpréteur de commande à utiliser (command.com sous Windows 95 ou cmd.exe sous Windows 2000 et XP).

Remarque : avec l'interpréteur de commande cmd.exe, il est nécessaire d'utiliser l'option /c qui permet de demander de quitter l'interpréteur à la fin de l'exécution de la commande.

L'inconvénient d'utiliser cette méthode est que la commande exécutée est dépendante du système d'exploitation.

La classe abstraite Process encapsule un processus : son implémentation est fournie par la JVM puisqu'elle est dépendante du système.

Les méthodes getOutputStream(), getInputStream() et getErrorStream() permettent d'avoir un accès respectivement au flux de sortie, d'entrée et d'erreur du processus.

La méthode waitFor() permet d'attendre la fin du processus

La méthode exitValue() permet d'obtenir le code retour du processus. Elle lève une exception de type IllegalThreadStateException si le processus n'est pas terminé.

La méthode destroy() permet de détruire le processus

Exemple :

```
package com.jmd.tests;

import java.io.BufferedReader;
import java.io.InputStreamReader;

public class TestRuntime2 {
    public static void main(String[] args) {
        try {
            Process proc =
                Runtime.getRuntime().exec("cmd.exe /c set");
            BufferedReader in =
                new BufferedReader(new InputStreamReader(proc.getInputStream()));
            String str;
            while ((str = in.readLine()) != null) {
                System.out.println(str);
            }
            in.close();
            proc.waitFor();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Cet exemple est fourni à titre d'exemple mais n'est pas solution idéale même si il fonctionne. Il est préférable de traiter les flux dans un thread dédié.

Exemple :

```
package com.jmd.tests;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;

public class TestRuntime3 {

  public TestRuntime3() {
    try {

      Runtime runtime = Runtime.getRuntime();
      Process proc = runtime.exec("cmd.exe /c set");

      TestRuntime3.AfficheFlux afficheFlux =
        new AfficheFlux(proc.getInputStream());

      afficheFlux.start();

      int exitVal = proc.waitFor();
      System.out.println("exitVal = " + exitVal);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  public static void main(String[] args) {
    new TestRuntime3();
  }

  private class AfficheFlux extends Thread {
    InputStream is;

    AfficheFlux(InputStream is) {
      this.is = is;
    }

    public void run() {
      try {
        InputStreamReader isr = new InputStreamReader(is);
        BufferedReader br = new BufferedReader(isr);
        String line = null;
        while ((line = br.readLine()) != null)
          System.out.println(line);
      } catch (IOException ioe) {
        ioe.printStackTrace();
      }
    }
  }
}
```

Sous Windows, il est possible d'utiliser un fichier dont l'extension est associée à une application

Exemple :

```
Process proc = Runtime.getRuntime().exec("cmd.exe /c \"c:\\test.doc\"");
```

5.3. La présentation rapide du package awt java

AWT est une collection de classes pour la réalisation d'applications graphiques ou GUI (Graphic User Interface)

Les composants qui sont utilisés par les classes définies dans ce package sont des composants dit "lourds" : ils dépendent entièrement du système d'exploitation. D'ailleurs leur nombre est limité car ils sont communs à plusieurs systèmes d'exploitation pour assurer la portabilité. Cependant, la représentation d'une interface graphique avec awt sur plusieurs systèmes peut ne pas être identique.

AWT se compose de plusieurs packages dont les principaux sont:

- java.awt : c'est le package de base de la bibliothèque AWT
- java.awt.images : ce package permet la gestion des images
- java.awt.event : ce package permet la gestion des événements utilisateurs
- java.awt.font : ce package permet d'utiliser les polices de caractères
- java.axt.dnd : ce package permet l'utilisation du cliquer/glisser

Le chapitre «La création d'interfaces graphiques avec AWT» détaille l'utilisation de ce package.

5.4. La présentation rapide du package java.io

Ce package définit un ensemble de classes pour la gestion des flux d'entrées-sorties.

Le chapitre «Les flux» détaille l'utilisation de ce package.

5.5. Le package java.util

Ce package contient un ensemble de classes utilitaires : la gestion des dates (Date et Calendar), la génération de nombres aléatoires (Random), la gestion des collections ordonnées ou non tel que la table de hachage (HashTable), le vecteur (Vector), la pile (Stack) ..., la gestion des propriétés (Properties), des classes dédiées à l'internationalisation (ResourceBundle, PropertyResourceBundle, ListResourceBundle) etc ...

Certaines de ces classes sont présentées plus en détail dans les sections suivantes.

5.5.1. La classe StringTokenizer

Cette classe permet de découper une chaîne de caractères (objet de type String) en fonction de séparateurs. Le constructeur de la classe accepte 2 paramètres : la chaîne à décomposer et une chaîne contenant les séparateurs

Exemple (code Java 1.1) :

```
import java.util.*;

class test9 {
   public static void main(String args[]) {
      StringTokenizer st = new StringTokenizer("chaine1,chaine2,chaine3,chaine4",",");
      while (st.hasMoreTokens()) {
         System.out.println((st.nextToken()).toString());
      }
   }
}

C:\java>java test9
chaine1
chaine2
chaine3
chaine4
```

La méthode hasMoreTokens() fournit un contrôle d'itération sur la collection en renvoyant un booléen indiquant si il reste encore des éléments.

La méthode getNextTokens() renvoie le prochain élément sous la forme d'un objet String

5.5.2. La classe Random

La classe Random permet de générer des nombres pseudo-aléatoires. Après l'appel au constructeur, il suffit d'appeler la méthode correspondant au type désiré : nextInt(), nextLong(), nextFloat() ou nextDouble()

Méthodes	valeur de retour
nextInt()	entre Integer.MIN_VALUE et Interger.MAX_VALUE
nextLong()	entre long.MIN_VALUE et long.MAX_VALUE
nextFloat() ou nextDouble()	entre 0.0 et 1.0

Exemple (code Java 1.1) :

```
import java.util.*;
class test9 {
   public static void main (String args[]) {
      Random r = new Random();
      int a = r.nextInt() %10; //entier entre -9 et 9
      System.out.println("a = "+a);
   }
}
```

5.5.3. Les classes Date et Calendar

En Java 1.0, la classe Date permet de manipuler les dates.

Exemple (code Java 1.0) :

```
import java.util.*;
...
    Date maintenant = new Date();
    if (maintenant.getDay() == 1)
       System.out.println(" lundi ");
```

Le constructeur d'un objet Date l'initialise avec la date et l'heure courante du système.

Exemple (code Java 1.0) :

```
import java.util.*;
import java.text.*;

public class TestHeure {
   public static void main(java.lang.String[] args) {
      Date date = new Date();
      System.out.println(DateFormat.getTimeInstance().format(date));
   }
}
```

Résultat :

```
22:05:21
```

La méthode getTime() permet de calculer le nombre de millisecondes écoulées entre la date qui est encapsulée dans l'objet qui reçoit le message getTime et le premier janvier 1970 à minuit GMT.

En java 1.1, de nombreuses méthodes et constructeurs de la classe Date sont deprecated, notamment celles qui permettent de manipuler les éléments qui composent la date et leur formatage : il faut utiliser la classe Calendar.

Exemple (code Java 1.1) :

```
import java.util.*;

public class TestCalendar {
   public static void main(java.lang.String[] args) {

      Calendar c = Calendar.getInstance();
      if (c.get(Calendar.DAY_OF_WEEK) == Calendar.MONDAY)
          System.out.println(" nous sommes lundi ");
      if (c.get(Calendar.DAY_OF_WEEK) == Calendar.TUESDAY)
          System.out.println(" nous sommes mardi ");
      if (c.get(Calendar.DAY_OF_WEEK) == Calendar.WEDNESDAY)
          System.out.println(" nous sommes mercredi ");
      if (c.get(Calendar.DAY_OF_WEEK) == Calendar.THURSDAY)
          System.out.println(" nous sommes jeudi ");
      if (c.get(Calendar.DAY_OF_WEEK) == Calendar.FRIDAY)
          System.out.println(" nous sommes vendredi ");
      if (c.get(Calendar.DAY_OF_WEEK) == Calendar.SATURDAY)
          System.out.println(" nous sommes samedi ");
      if (c.get(Calendar.DAY_OF_WEEK) == Calendar.SUNDAY)
          System.out.println(" nous sommes dimanche ");

   }
}
```

Résultat :

```
 nous sommes lundi
```

5.5.4. La classe SimpleDateFormat

La classe SimpleDateFormat permet de formater et d'analyser une date en tenant compte d'une Locale. Elle hérite de la classe abstraite DateFormat.

Pour réaliser ces traitements cette classe utilise un modèle (pattern) sous la forme d'une chaîne de caractères.

La classe DataFormat propose plusieurs méthodes pour obtenir le modèle par défaut de la Locale courante :

- getTimeInstance(style)
- getDateInstance(style)
- getDateTimeInstance(styleDate, styleHeure)

Ces méthodes utilise la Locale par défaut mais chacune de ces méthodes possède une surcharge qui permet de préciser une Locale.

Pour chacune de ces méthodes, quatre styles sont utilisables : SHORT, MEDIUM, LONG et FULL. Ils permettent de désigner la richesse des informations contenues dans le modèle pour la date et/ou l'heure.

Exemple :

```java
package com.jmd.test.dej.date;

import java.text.DateFormat;
import java.util.Date;
import java.util.Locale;

public class TestFormaterDate2 {

  /**
   * @param args
   */
  public static void main(String[] args) {
    Date aujourdhui = new Date();

    DateFormat shortDateFormat = DateFormat.getDateTimeInstance(
        DateFormat.SHORT,
        DateFormat.SHORT);

    DateFormat shortDateFormatEN = DateFormat.getDateTimeInstance(
        DateFormat.SHORT,
        DateFormat.SHORT, new Locale("EN","en"));

    DateFormat mediumDateFormat = DateFormat.getDateTimeInstance(
        DateFormat.MEDIUM,
        DateFormat.MEDIUM);

    DateFormat mediumDateFormatEN = DateFormat.getDateTimeInstance(
        DateFormat.MEDIUM,
        DateFormat.MEDIUM, new Locale("EN","en"));

    DateFormat longDateFormat = DateFormat.getDateTimeInstance(
        DateFormat.LONG,
        DateFormat.LONG);

    DateFormat longDateFormatEN = DateFormat.getDateTimeInstance(
        DateFormat.LONG,
        DateFormat.LONG, new Locale("EN","en"));

    DateFormat fullDateFormat = DateFormat.getDateTimeInstance(
        DateFormat.FULL,
        DateFormat.FULL);

    DateFormat fullDateFormatEN = DateFormat.getDateTimeInstance(
        DateFormat.FULL,
        DateFormat.FULL, new Locale("EN","en"));

    System.out.println(shortDateFormat.format(aujourdhui));
    System.out.println(mediumDateFormat.format(aujourdhui));
    System.out.println(longDateFormat.format(aujourdhui));
    System.out.println(fullDateFormat.format(aujourdhui));
    System.out.println("");
    System.out.println(shortDateFormatEN.format(aujourdhui));
    System.out.println(mediumDateFormatEN.format(aujourdhui));
    System.out.println(longDateFormatEN.format(aujourdhui));
    System.out.println(fullDateFormatEN.format(aujourdhui));
  }

}
```

Résultat :

```
27/06/06 21:36
27 juin 2006 21:36:30
27 juin 2006 21:36:30 CEST
mardi 27 juin 2006 21 h 36 CEST

6/27/06 9:36 PM
Jun 27, 2006 9:36:30 PM
June 27, 2006 9:36:30 PM CEST
Tuesday, June 27, 2006 9:36:30 PM CEST
```

Il est aussi possible de définir son propre format en utilisant les éléments du tableau ci-dessous. Chaque lettre du tableau est interprétée de façon particulière. Pour utiliser les caractères sans qu'ils soient interprétés dans le modèle il faut les encadrer par de simples quotes. Pour utiliser une quote il faut en mettre deux consécutives dans le modèle.

Lettre	*Description*	*Exemple*
G	Era	AD (Anno Domini), BC (Before Christ)
y	Année	06 ; 2006
M	Mois dans l'année	Septembre; Sept.; 07
w	Semaine dans l'année	34
W	Semaine dans le mois	2
D	Jour dans l'année	192
d	jour dans le mois	23
F	Jour de la semaine dans le mois	17
E	Jour de la semaine	Mercredi; Mer.
a	Marqueur AM/PM (Ante/Post Meridiem)	PM, AM
H	Heure (0-23)	23
k	Heure (1-24)	24
K	Heure en AM/PM (0-11)	6
h	Heure en AM/PM (1-12)	7
m	Minutes	59
s	Secondes	59
S	Millisecondes	12564
z	Zone horaire générale	CEST; Heure d'été d'Europe centrale
Z	Zone horaire (RFC 822)	+0200

Ces caractères peuvent être répétés pour préciser le format à utiliser :

- Pour les caractères de type Text : moins de 4 caractères consécutifs représentent la version abrégée sinon c'est la version longue qui est utilisée.
- Pour les caractères de type Number : c'est le nombre de répétitions qui désigne le nombre de chiffre utilisé complété si nécessaire par des 0 à gauche.
- Pour les caractères de type Year : 2 caractères précisent que l'année est codée sur deux caractères.
- Pour les caractères de type Month : 3 caractères ou plus représentent la forme littérale sinon c'est la forme numérique du mois.

Exemple :

```
package com.jmd.test.dej.date;

import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Locale;

public class TestFormaterDate {

  public static void main(String[] args) {
    SimpleDateFormat formater = null;

    Date aujourdhui = new Date();

    formater = new SimpleDateFormat("dd-MM-yy");
```

```
    System.out.println(formater.format(aujourdhui));

    formater = new SimpleDateFormat("ddMMyy");
    System.out.println(formater.format(aujourdhui));

    formater = new SimpleDateFormat("yyMMdd");
    System.out.println(formater.format(aujourdhui));

    formater = new SimpleDateFormat("h:mm a");
    System.out.println(formater.format(aujourdhui));

    formater = new SimpleDateFormat("K:mm a, z");
    System.out.println(formater.format(aujourdhui));

    formater = new SimpleDateFormat("hh:mm a, zzzz");
    System.out.println(formater.format(aujourdhui));

    formater = new SimpleDateFormat("EEEE, d MMM yyyy");
    System.out.println(formater.format(aujourdhui));

    formater = new SimpleDateFormat("'le' dd/MM/yyyy 'à' hh:mm:ss");
    System.out.println(formater.format(aujourdhui));

    formater = new SimpleDateFormat("'le' dd MMMM yyyy 'à' hh:mm:ss");
    System.out.println(formater.format(aujourdhui));

    formater = new SimpleDateFormat("dd MMMMM yyyy GGG, hh:mm aaa");
    System.out.println(formater.format(aujourdhui));

    formater = new SimpleDateFormat("yyyyMMddHHmmss");
    System.out.println(formater.format(aujourdhui));

  }

}
```

Résultat :

```
27-06-06
270606
060627
9:37 PM
9:37 PM, CEST
09:37 PM, Heure d'été d'Europe centrale
mardi, 27 juin 2006
le 27/06/2006 à 09:37:10
le 27 juin 2006 à 09:37:10
27 juin 2006 ap. J.-C., 09:37 PM
20060627213710
```

Il existe plusieurs constructeurs de la classe SimpleDateFormat :

Constructeur	Rôle
SimpleDateFormat()	Constructeur par défaut utilisant le modèle par défaut et les symboles de formatage de dates de la Locale par défaut
SimpleDateFormat(String)	Constructeur utilisant le modèle fourni et les symboles de formatage de dates de la Locale par défaut
SimpleDateFormat(String, DateFormatSymbols)	Constructeur utilisant le modèle et les symboles de formatage de dates fournis
SimpleDateFormat(String, Locale)	Constructeur utilisant le modèle fourni et les symboles de formatage de dates de la Locale fournie

La classe DateFormatSymbols encapsule les différents éléments textuels qui peuvent entrer dans la composition d'une date pour une Locale donnée (les jours, les libellés courts des mois, les libellés des mois, ...).

Exemple :

```
package com.jmd.test.dej.date;

import java.text.DateFormatSymbols;
import java.util.Locale;

public class TestFormaterDate3 {

  public static void main(String[] args) {
    DateFormatSymbols dfsFR = new DateFormatSymbols(Locale.FRENCH);
    DateFormatSymbols dfsEN = new DateFormatSymbols(Locale.ENGLISH);

    String[] joursSemaineFR = dfsFR.getWeekdays();
    String[] joursSemaineEN = dfsEN.getWeekdays();

    StringBuffer texteFR = new StringBuffer("Jours FR ");
    StringBuffer texteEN = new StringBuffer("Jours EN ");

    for (int i = 1; i < joursSemaineFR.length; i++) {
      texteFR.append(" : ");
      texteFR.append(joursSemaineFR[i]);
      texteEN.append(" : ");
      texteEN.append(joursSemaineEN[i]);
    }
    System.out.println(texteFR);
    System.out.println(texteEN);

    texteFR = new StringBuffer("Mois courts FR ");
    texteEN = new StringBuffer("Mois courts EN ");
    String[] moisCourtsFR = dfsFR.getShortMonths();
    String[] moisCourtsEN = dfsEN.getShortMonths();

    for (int i = 0; i < moisCourtsFR.length - 1; i++) {
      texteFR.append(" : ");
      texteFR.append(moisCourtsFR[i]);
      texteEN.append(" : ");
      texteEN.append(moisCourtsEN[i]);
    }

    System.out.println(texteFR);
    System.out.println(texteEN);

    texteFR = new StringBuffer("Mois FR ");
    texteEN = new StringBuffer("Mois EN ");
    String[] moisFR = dfsFR.getMonths();
    String[] moisEN = dfsEN.getMonths();

    for (int i = 0; i < moisFR.length - 1; i++) {
      texteFR.append(" : ");
      texteFR.append(moisFR[i]);
      texteEN.append(" : ");
      texteEN.append(moisEN[i]);
    }

    System.out.println(texteFR);
    System.out.println(texteEN);

  }

}
```

Résultat :

```
Jours FR  : dimanche : lundi : mardi : mercredi : jeudi : vendredi : samedi
Jours EN  : Sunday : Monday : Tuesday : Wednesday : Thursday : Friday : Saturday
Mois courts FR  : janv. : févr. : mars : avr. : mai : juin : juil. : août : sept. : oct.
 : nov. : déc.
Mois courts EN  : Jan : Feb : Mar : Apr : May : Jun : Jul : Aug : Sep : Oct : Nov : Dec
Mois FR  : janvier : février : mars : avril : mai : juin : juillet : août : septembre :
octobre : novembre : décembre
Mois EN  : January : February : March : April : May : June : July : August : September :
 October : November : December
```

Il est possible de définir son propre objet DateFormatSymbols pour personnaliser les éléments textuels nécessaires pour le traitement des dates. La classe DateFormatSymbols propose à cet effet des setters sur chacun des éléments.

Exemple :

```
package com.jmd.test.dej.date;

import java.text.DateFormatSymbols;
import java.text.SimpleDateFormat;
import java.util.Date;

public class TestFormaterDate4 {

  public static void main(String[] args) {
    Date aujourdhui = new Date();
    DateFormatSymbols monDFS = new DateFormatSymbols();
    String[] joursCourts = new String[] {
        "",
        "Di",
        "Lu",
        "Ma",
        "Me",
        "Je",
        "Ve",
        "Sa" };
    monDFS.setShortWeekdays(joursCourts);
    SimpleDateFormat dateFormat = new SimpleDateFormat(
        "EEE dd MMM yyyy HH:mm:ss",
        monDFS);
    System.out.println(dateFormat.format(aujourdhui));
  }

}
```

Résultat :

```
Ma 27 juin 2006 21:38:22
```

Attention : il faut consulter la documentation de l'API pour connaître précisément le contenu et l'ordre des éléments fournis sous la forme de tableau au setter de la classe. Dans l'exemple, ci-dessus, les jours de la semaine commencent par dimanche.

La méthode applyPattern() permet de modifier le modèle d'un objet SimpleDateFormat.

La classe SimpleDataFormat permet également d'analyser une date sous la forme d'une chaîne de caractères pour la transformer en objet de type Date en utilisant un modèle. Cette opération est réalisée grâce à la méthode parse(). Si elle échoue, elle lève une exception de type ParseException.

Exemple :

```
package com.jmd.test.dej.date;

import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

public class TestParserDate {

  public static void main(String[] args) {
    Date date = null;
    SimpleDateFormat simpleDateFormat = new SimpleDateFormat("dd/MM/yyyy");

    String date1 = "22/06/2006";
    String date2 = "22062006";

    try {
```

```
      date = simpleDateFormat.parse(date1);
      System.out.println(date);
      date = simpleDateFormat.parse(date2);
      System.out.println(date);
    } catch (ParseException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
Thu Jun 22 00:00:00 CEST 2006
java.text.ParseException: Unparseable date: "22062006"
        at java.text.DateFormat.parse(Unknown Source)
        at com.jmd.test.dej.date.TestParserDate.main(TestParserDate.java:19)
```

5.5.5. La classe Vector

Un objet de la classe Vector peut être considéré comme une tableau évolué qui peut contenir un nombre indéterminé d'objets.

Les méthodes principales sont les suivantes :

Méthode	Rôle
void addElement(Object)	ajouter un objet dans le vecteur
boolean contains(Object)	retourne true si l'objet est dans le vecteur
Object elementAt(int)	retourne l'objet à l'index indiqué
Enumeration elements()	retourne une énumeration contenant tous les éléments du vecteur
Object firstElement()	retourne le premier élément du vecteur (celui dont l'index est égal à zéro)
int indexOf(Object)	renvoie le rang de l'élément ou -1
void insertElementAt(Object, int)	insérer un objet à l'index indiqué
boolean isEmpty()	retourne un booléen si le vecteur est vide
Objet lastElement()	retourne le dernier élément du vecteur
void removeAllElements()	vider le vecteur
void removeElement(Object)	supprime l'objet du vecteur
void removeElementAt(int)	supprime l'objet à l'index indiqué
void setElementAt(object, int)	remplacer l'élément à l'index par l'objet
int size()	nombre d'objet du vecteur

On peut stocker des objets de classes différentes dans un vecteur mais les éléments stockés doivent obligatoirement être des objets (pour le type primitif il faut utiliser les wrappers tel que Integer ou Float mais pas int ou float).

Exemple (code Java 1.1) :

```
Vector v = new Vector();
v.addElement(new Integer(10));
v.addElement(new Float(3.1416));
v.insertElementAt("chaine ",1);
System.out.println(" le vecteur contient "+v.size()+ " elements ");
```

```
String retrouve = (String) v.elementAt(1);
System.out.println(" le 1er element = "+retrouve);

C:\$user\java>java test9
le vecteur contient 3 elements
le 1er element = chaine
```

Exemple (code Java 1.1) :

```
   Vector v = new Vector();
   ...

   for (int i = 0; i < v.size() ; i ++) {
      System.out.println(v.elementAt(i));
   }
```

Il est aussi possible de parcourir l'ensemble des éléments en utilisant une instance de l'interface Enumeration.

5.5.6. La classe Hashtable

Les informations d'une Hastable sont stockées sous la forme clé - données. Cet objet peut être considéré comme un dictionnaire.

Exemple (code Java 1.1) :

```
Hashtable dico = new Hashtable();
dico.put("livre1", " titre du livre 1 ");
dico.put("livre2", "titre du livre 2 ");
```

Il est possible d'utiliser n'importe quel objet comme clé et comme donnée

Exemple (code Java 1.1) :

```
dico.put("jour", new Date());
dico.put(new Integer(1),"premier");
dico.put(new Integer(2),"deuxième");
```

Pour lire dans la table, on utilise get(object) en donnant la clé en paramètre.

Exemple (code Java 1.1) :

```
System.out.println(" nous sommes le " +dico.get("jour"));
```

La méthode remove(Object) permet de supprimer une entrée du dictionnaire correspondant à la clé passée en paramètre.

La méthode size() permet de connaitre le nombre d'association du dictionnaire.

5.5.7. L'interface Enumeration

L'interface Enumeration est utilisée pour permettre le parcours séquentiel de collections.

Enumeration est une interface qui définit 2 méthodes :

Méthodes	Rôle

boolean hasMoreElements()	retourne true si l'énumération contient encore un ou plusieurs éléments
Object nextElement()	retourne l'objet suivant de l'énumération Elle lève une Exception NoSuchElementException si la fin de la collection est atteinte.

Exemple (code Java 1.1) : contenu d'un vecteur et liste des clés d'une Hastable

```
import java.util.*;

class test9 {

   public static void main (String args[]) {

      Hashtable h = new Hashtable();
      Vector v = new Vector();

      v.add("chaine 1");
      v.add("chaine 2");
      v.add("chaine 3");

      h.put("jour", new Date());
      h.put(new Integer(1),"premier");
      h.put(new Integer(2),"deuxième");

      System.out.println("Contenu du vector");

      for (Enumeration e = v.elements() ; e.hasMoreElements() ; ) {
         System.out.println(e.nextElement());
      }

      System.out.println("\nContenu de la hashtable");

      for (Enumeration e = h.keys() ; e.hasMoreElements() ; ) {
         System.out.println(e.nextElement());
      }
   }
}

C:\$user\java>java test9
Contenu du vector
chaine 1
chaine 2
chaine 3
Contenu de la hashtable
jour
2
1
```

5.5.8. La manipulation d'archives zip

Depuis sa version 1.1, le JDK propose des classes permettant la manipulation d'archives au format zip. Ce format de compression est utilisé par Java lui-même notamment pour les fichiers de packaging (jar, war, ear ...).

Ces classes sont regroupées dans le package java.util.zip. Elles permettent de manipuler les archives au format zip et Gzip et d'utiliser des sommes de contrôles selon les algorithmes Adler-32 et CRC-32.

La classe ZipFile encapsule une archive au format zip : elle permet de manipuler les entrées qui composent l'archive.

Elle possède trois constructeurs :

Constructeur	Rôle

ZipFile(File)	ouvre l'archive correspondant au fichier fourni en paramètre
ZipFile(File, int)	ouvre l'archive correspondant au fichier fourni en paramètre selon le mode précisé : OPEN_READ ou OPEN_READ | OPEN_DELETE
ZipFile(string)	ouvre l'archive dont le nom de fichier est fourni en paramètre

Exemple :

```
try {
 ZipFile test = new ZipFile(new File("C:/test.zip"));
}  catch (ZipException e) {
 e.printStackTrace();
}  catch (IOException e) {
 e.printStackTrace();
}
```

Cette classe possède plusieurs méthodes dont les principales sont :

Méthode	Rôle
Enumeration entries()	Obtenir une énumération des entrées de l'archive sous la forme d'objet de type ZipEntry
close()	Fermer l'archive
ZipEntry getEntry(String)	Renvoie l'entrée dont le nom est précisé en paramètre
InputStream getInputStream(ZipEntry)	Renvoie un flux de lecture pour l'entrée précisée

La classe ZipEntry encapsule une entrée dans l'archive zip. Une entrée correspond à un fichier avec des informations le concernant dans l'archive.

Cette classe possède plusieurs méthodes dont les principales sont :

Méthode	Rôle
getName()	Renvoie le nom de l'entrée (nom du fichier avec sa sous arborescence dans l'archive)
getTime() / setTime()	Renvoie ou modifie la date de modification de l'entrée
getComment() / setComment()	Renvoie ou modifie le commentaire associé à l'entrée
getSize() / setSize()	Renvoie ou modifie la taille de l'entrée non compressée
getCompressedSize() / setCompressedSize()	Renvoie ou modifie la taille de l'entrée compressée
getCrc() / setCrc()	Renvoie ou modifie la somme de contrôle permettant de vérifier l'intégrité de l'entrée
getMethod() / setMethod()	Renvoie ou modifie la méthode utilisée pour la compression
isDirectory()	Renvoie un booléen précisant si l'entrée est un répertoire

Exemple : afficher le contenu d'une archive

```
public static void listerZip(String nomFichier) {
  ZipFile zipFile;
  try {
    zipFile = new ZipFile(nomFichier);
    Enumeration entries = zipFile.entries();
    while (entries.hasMoreElements()) {
      ZipEntry entry = (ZipEntry) entries.nextElement();
```

```
        String name = entry.getName();
        System.out.println(name);
      }
      zipFile.close();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
```

La classe ZipOutputStream est un flux qui permet l'écriture de données dans l'archive.

Cette classe possède plusieurs méthodes dont les principales sont :

Méthode	Rôle
setMethod()	Modifier la méthode de compression utilisée par défaut. Les valeurs possibles sont STORED (aucune compression) ou DEFLATED (avec compression)
setLevel()	Modifier le taux de compression : les valeurs entières possibles sont 0 à 9 ou 9 correspond au taux de compression le plus élevé. Des constantes sont définies dans la classe Deflater : Deflater.BEST_COMPRESSION, Deflater.DEFAULT_COMPRESSION, Deflater.BEST_SPEED, Deflater.NO_COMPRESSION
putNextEntry(ZipEntry)	Permet de se positionner dans l'archive pour ajouter l'entrée fournie en paramètre
write(byte[] b, int off, int len)	Permet d'écrire un tableau d'octet dans l'entrée courante
closeEntry()	Fermer l'entrée courante et se positionne pour ajouter l'entrée suivante
close()	Fermer le flux

Exemple : compresser un fichier dans une archive

```
  public static void compresser(String nomArchive, String nomFichier) {
    try {
      ZipOutputStream zip = new ZipOutputStream(
          new FileOutputStream(nomArchive));
      zip.setMethod(ZipOutputStream.DEFLATED);
      zip.setLevel(Deflater.BEST_COMPRESSION);

      // lecture du fichier
      File fichier = new File(nomFichier);
      FileInputStream fis = new FileInputStream(fichier);
      byte[] bytes = new byte[fis.available()];
      fis.read(bytes);

      // ajout d'une nouvelle entree dans l'archive contenant le fichier
      ZipEntry entry = new ZipEntry(nomFichier);
      entry.setTime(fichier.lastModified());
      zip.putNextEntry(entry);
      zip.write(bytes);

      // fermeture des flux
      zip.closeEntry();
      fis.close();
      zip.close();
    } catch (FileNotFoundException fnfe) {
      fnfe.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }
  }
```

La classe ZipInputStream est un flux qui permet la lecture de données dans l'archive.

Cette classe possède plusieurs méthodes dont les principales sont :

Méthode	Rôle
getNextEntry()	Permet de se positionner sur l'entrée suivante dans l'archive
read(byte[] b, int off, int len)	permet de lire un tableau d'octets dans l'entrée courante
close()	permet de fermer le flux

Exemple :

```
public static void decompresser(String nomArchive, String chemin) {
  try {

    ZipFile zipFile = new ZipFile(nomArchive);
    Enumeration entries = zipFile.entries();
    ZipEntry entry = null;
    File fichier = null;
    File sousRep = null;

    while (entries.hasMoreElements()) {
      entry = (ZipEntry) entries.nextElement();

      if (!entry.isDirectory()) {
        System.out.println("Extraction du fichier " + entry.getName());
        fichier = new File(chemin + File.separatorChar + entry.getName());
        sousRep = fichier.getParentFile();

        if (sousRep != null) {
          if (!sousRep.exists()) {
            sousRep.mkdirs();
          }
        }

        int i = 0;
        byte[] bytes = new byte[1024];
        BufferedOutputStream out = new BufferedOutputStream(
            new FileOutputStream(fichier));
        BufferedInputStream in = new BufferedInputStream(zipFile
            .getInputStream(entry));
        while ((i = in.read(bytes)) != -1)
          out.write(
              bytes,
              0,
              i);

        in.close();
        out.flush();
        out.close();
      }
    }
    zipFile.close();
  } catch (FileNotFoundException fnfe) {
    fnfe.printStackTrace();
  } catch (IOException ioe) {
    ioe.printStackTrace();
  }
}
```

5.5.9. Les expressions régulières

Le JDK 1.4 propose une api en standard pour utiliser les expressions régulières. Les expressions régulières permettent de comparer une chaîne de caractères à un motif pour vérifier qu'il y a concordance.

La package java.util.regexp contient deux classes et une exception pour gérer les expressions régulières :

Classe	Rôle
Matcher	comparer une chaîne de caractère avec un motif
Pattern	encapsule une version compilée d'un motif
PatternSyntaxException	exception levée lorsque le motif contient une erreur de syntaxe

5.5.9.1. Les motifs

Les expressions régulières utilisent un motif. Ce motif est une chaîne de caractères qui contient des caractères et des méta caractères. Les méta caractères ont une signification particulière et sont interprétés.

Il est possible de déspécialiser un méta caractère (lui enlever sa signification particulière) en le faisant précéder d'un caractère backslash. Ainsi pour utiliser le caractère backslash, il faut le doubler.

Les méta caractères reconnus par l'api sont :

méta caractères	rôle
()	
[]	définir un ensemble de caractères
{}	définir une répétition du motif précédent
\	déspécialisation du caractère qui suit
^	début de la ligne
$	fin de la ligne
|	le motif précédent ou le motif suivant
?	motif précédent répété zéro ou une fois
*	motif précédent répété zéro ou plusieurs fois
+	motif précédent répété une ou plusieurs fois
.	un caractère quelconque

Certains caractères spéciaux ont une notation particulière :

Notation	Rôle
\t	tabulation
\n	nouvelle ligne (ligne feed)
\\	backslash

Il est possible de définir des ensembles de caractères à l'aide des caractères [et]. Il suffit d'indiquer les caractères de l'ensemble entre ces deux caractères

Exemple : toutes les voyelles

```
[aeiouy]
```

Il est possible d'utiliser une plage de caractères consécutifs en séparant le caractère de début de la plage et le caractère de fin de la plage avec un caractère -

Exemple : toutes les lettres minuscules
[a-z]

L'ensemble peut être l'union de plusieurs plages.

Exemple : toutes les lettres
[a-zA-Z]

Par défaut l'ensemble [] désigne tous les caractères. Il est possible de définir un ensemble de la forme tous sauf ceux précisés en utilisant le caractère ^ suivi des caractères à enlever de l'ensemble

Exemple : tous les caractères sauf les lettres
[^a-zA-Z]

Il existe plusieurs ensembles de caractères prédéfinis :

Notation	Contenu de l'ensemble
\d	un chiffre
\D	tous sauf un chiffre
\w	une lettre ou un underscore
\W	tous sauf une lettre ou un underscore
\s	un séparateur (espace, tabulation, retour chariot, ...)
\S	tous sauf un séparateur

Plusieurs méta caractères permettent de préciser un critère de répétition d'un motif

méta caractères	rôle
{n}	répétition du motif précédent n fois
{n,m}	répétition du motif précédent entre n et m fois
{n,}	répétition du motif précédent
?	motif précédent répété zéro ou une fois
*	motif précédent repété zéro ou plusieurs fois
+	motif précédent répété une ou plusieurs fois

Exemple : la chaîne AAAAA
A{5}

5.5.9.2. La classe Pattern

Cette classe encapsule une représentation compilée d'un motif d'une expression régulière.

La classe Pattern ne possède pas de constructeur public mais propose une méthode statique compile().

Exemple :

```
private static Pattern motif = null;
...
motif = Pattern.compile("liste[0-9]");
```

Une version surchargée de la méthode compile() permet de préciser certaines options dont la plus intéressante permet de rendre insensible à la casse les traitements en utilisant le flag CASE_INSENSITIVE.

Exemple :

```
private static Pattern motif = null;
...
motif = Pattern.compile("liste[0-9]",Pattern.CASE_INSENSITIVE);
```

Cette méthode compile() renvoie une instance de la classe Pattern si le motif est syntaxiquement correcte sinon elle lève une exception de type PatternSyntaxException.

La méthode matches(String, String) permet de rapidement et facilement utiliser les expressions régulières avec un seul appel de méthode en fournissant le motif est la chaîne à traiter.

Exemple :

```
if (Pattern.matches("liste[0-9]","liste2")) {
      System.out.println("liste2 ok");
} else {
      System.out.println("liste2 ko");
}
```

5.5.9.3. La classe Matcher

La classe Matcher est utilisée pour effectuer la comparaison entre une chaîne de caractères et un motif encapsulé dans un objet de type Pattern.

Cette classe ne possède aucun constructeur public. Pour obtenir une instance de cette classe, il faut utiliser la méthode matcher() d'une instance d'un objet Pattern en lui fournissant la chaîne à traiter en paramètre.

Exemple :

```
motif = Pattern.compile("liste[0-9]");
matcher = motif.matcher("liste1");
```

Exemple :

```
motif = Pattern.compile("liste[0-9]");
matcher = motif.matcher("liste1");
if (matcher.matches()) {
      System.out.println("liste1 ok");
} else {
      System.out.println("liste1 ko");
}
matcher = motif.matcher("liste10");
if (matcher.matches()) {
      System.out.println("liste10 ok");
} else {
      System.out.println("liste10 ko");
}
```

Résultat :

```
liste1 ok
liste10 ko
```

La méthode lookingAt() tente de recherche le motif dans la chaîne à traiter

Exemple :

```
            motif = Pattern.compile("liste[0-9]");
            matcher = motif.matcher("liste1");
            if (matcher.lookingAt()) {
                  System.out.println("liste1 ok");
            } else {
                  System.out.println("liste1 ko");
            }
            matcher = motif.matcher("liste10");
            if (matcher.lookingAt()) {
                  System.out.println("liste10 ok");
            } else {
                  System.out.println("liste10 ko");
            }
```

Résultat :

```
 liste1 ok
 liste10 ok
```

La méthode find() permet d'obtenir des informations sur chaque occurrence ou le motif est trouvé dans la chaîne à traiter.

Exemple :

```
      matcher = motif.matcher("zzliste1zz");
      if (matcher.find()) {
            System.out.println("zzliste1zz ok");
      } else {
           System.out.println("zzliste1zz ko");
      }
```

Résultat :

```
zzliste1zz ok
```

Il est possible d'appeler successivement cette méthode pour obtenir chacune des occurrences.

Exemple :

```
        int i = 0;
        motif = Pattern.compile("liste[0-9]");
        matcher = motif.matcher("liste1liste2liste3");
        while (matcher.find()) {
              i++;
        }
        System.out.println("nb occurrences = " + i);
```

Les méthodes start() et end() permettent de connaitre la position de début et de fin dans la chaîne dans l'occurrence en cours de traitement.

Exemple :

```
        int i = 0;
        motif = Pattern.compile("liste[0-9]");
        matcher = motif.matcher("liste1liste2liste3");
        while (matcher.find()) {
```

```
            System.out.print("pos debut : "+matcher.start());
            System.out.println(" pos fin : "+matcher.end());
            i++;
        }
        System.out.println("nb occurrences = " + i);
```

Résultat :

```
  pos debut : 0 pos fin : 6
  pos debut : 6 pos fin : 12
  pos debut : 12 pos fin : 18
  nb occurences = 3
```

La classe Matcher propose aussi les méthodes replaceFirst() et replaceAll() pour facilement remplacer la première ou toutes les occurrences du motif trouvé par une chaîne de caractères.

Exemple : remplacement de la première occurrence

```
        motif = Pattern.compile("liste[0-9]");
        matcher = motif.matcher("zz liste1 zz liste2 zz");
        System.out.println(matcher.replaceFirst("chaine"));
```

Résultat :

```
zz chaine zz liste2 zz
```

Exemple : remplacement de toutes les occurrences

```
      motif = Pattern.compile("liste[0-9]");
      matcher = motif.matcher("zz liste1 zz liste2 zz");
      System.out.println(matcher.replaceAll("chaine"));
```

Résultat :

```
zz chaine zz chaine zz
```

5.5.10. La classe Formatter

La méthode printf() utilise la classe Formatter pour réaliser le formatage des données fournies selon leurs valeurs et le format donné en paramètre.

Cette classe peut aussi être utilisée pour formater des données pour des fichiers ou dans une servlet par exemple.

La méthode format() attend en paramètre une chaîne de caractères qui précise le format les données à formater.

Exemple (java 1.5) :

```
import java.util.*;

public class TestFormatter {

  public static void main(String[] args) {
    Formatter formatter = new Formatter();
    formatter.format("%04d \n",13);
    String resultat = formatter.toString();
    System.out.println("chaine = " + resultat);

  }

}
```

Résultat :

```
C:\tiger>java TestFormatter
chaine = 0013
```

5.5.11. La classe Scanner

Cette classe facilite la lecture dans un flux. Elle est particulièrement utile pour réaliser une lecture de données à partir du clavier dans une application de type console.

La méthode next() bloque l'exécution jusqu'à la lecture de données et les renvoie sous la forme d'une chaîne de caractères.

Exemple (java 1.5) :

```
import java.util.*;

public class TestScanner {

  public static void main(String[] args) {
    Scanner scanner = Scanner.create(System.in);
    String chaine = scanner.next();
    scanner.close();

  }

}
```

Cette classe possède plusieurs méthodes nextXXX() ou XXX représente un type primitif. Ces méthodes bloquent l'exécution jusqu'à la lecture de données et tente de les convertir dans le type XXX

Exemple (java 1.5) :

```
import java.util.*;

public class TestScanner {

  public static void main(String[] args) {
    Scanner scanner = Scanner.create(System.in);
    int entier = scanner.nextInt();
    scanner.close();
  }

}
```

Une exception de type InputMismatchException est levée si les données lue dans le flux ne sont pas du type requis.

Exemple (java 1.5) :

```
C:\tiger>java TestScanner
texte
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Unknown Source)
        at java.util.Scanner.next(Unknown Source)
        at java.util.Scanner.nextInt(Unknown Source)
        at java.util.Scanner.nextInt(Unknown Source)
        at TestScanner.main(TestScanner.java:8)
```

La classe Scanner peut être utilisée avec n'importe quel flux.

5.6. La présentation rapide du package java.net

Ce package contient un ensemble de classes pour permettre une interaction avec le réseau pour permettre de recevoir et d'envoyer des données à travers ce dernier.

Le chapitre «L'interaction avec le réseau» détaille l'utilisation de ce package.

5.7. La présentation rapide du package java.applet

Ce package contient les classes nécessaires au développement des applets. Une applet est une petite application téléchargée par le réseau et exécutée sous de fortes contraintes de sécurité dans une page Web par le navigateur.

Le développement des applets est détaillé dans le chapitre «Les applets en java»

6. Les fonctions mathématiques

La classe java.lang.Math contient une série de méthodes et variables mathématiques. Comme la classe Math fait partie du package java.lang, elle est automatiquement importée. De plus, il n'est pas nécessaire de déclarer un objet de type Math car les méthodes sont toutes static

Exemple (code Java 1.1) : Calculer et afficher la racine carrée de 3

```
public class Math1 {
  public static void main(java.lang.String[] args) {
    System.out.println(" = " + Math.sqrt(3.0));
  }
}
```

Ce chapitre contient plusieurs sections :

- Les variables de classe
- Les fonctions trigonométriques
- Les fonctions de comparaisons
- Les arrondis
- La méthode IEEEremainder(double, double)
- Les Exponentielles et puissances
- La génération de nombres aléatoires
- La classe BigDecimal

6.1. Les variables de classe

PI représente pi dans le type double (3,14159265358979323846)

E représente e dans le type double (2,7182818284590452354)

Exemple (code Java 1.1) :

```
public class Math2 {
  public static void main(java.lang.String[] args) {
    System.out.println(" PI = "+Math.PI);
    System.out.println(" E = "+Math.E);
  }
}
```

6.2. Les fonctions trigonométriques

Les méthodes sin(), cos(), tan(), asin(), acos(), atan() sont déclarées : public static double fonctiontrigo(double angle)

Les angles doivent être exprimés en radians. Pour convertir des degrés en radian, il suffit de les multiplier par PI/180

6.3. Les fonctions de comparaisons

max (n1, n2)
min (n1, n2)

Ces méthodes existent pour les types int, long, float et double : elles déterminent respectivement les valeurs maximales et minimales des deux paramètres.

Exemple (code Java 1.1) :

```
public class Math1 {
        public static void main(String[] args) {
                System.out.println(" le plus grand = " + Math.max(5, 10));
                System.out.println(" le plus petit = " + Math.min(7, 14));
        }
}
```

Résultat :

```
le plus grand = 10
le plus petit = 7
```

6.4. Les arrondis

La classe Math propose plusieurs méthodes pour réaliser différents arrondis.

6.4.1. La méthode round(n)

Cette méthode ajoute 0,5 à l'argument et restitue la plus grande valeur entière (int) inférieure ou égale au résultat. La méthode est définie pour les types float et double.

Exemple (code Java 1.1) :

```
public class Arrondis1 {
  static double[] valeur = {-5.7, -5.5, -5.2, -5.0, 5.0, 5.2, 5.5, 5.7 };

  public static void main(String[] args) {
    for (int i = 0; i < valeur.length; i++) {
      System.out.println("round("+valeur[i]+") = "+Math.round(valeur[i]));
    }
  }
}
```

Résultat :

```
round(-5.7) = -6
round(-5.5) = -5
round(-5.2) = -5
round(-5.0) = -5
round(5.0) = 5
round(5.2) = 5
```

```
round(5.5) = 6
round(5.7) = 6
```

6.4.2. La méthode rint(double)

Cette méthode effectue la même opération mais renvoie un type double.

Exemple (code Java 1.1) :

```
public class Arrondis2 {
  static double[] valeur = {-5.7, -5.5, -5.2, -5.0, 5.0, 5.2, 5.5, 5.7 };

  public static void main(String[] args) {
    for (int i = 0; i > valeur.length; i++) {
    System.out.println("rint("+valeur[i]+") = "+Math.rint(valeur[i]));
    }
  }
}
```

Résultat :

```
rint(-5.7) = -6.0
rint(-5.5) = -6.0
rint(-5.2) = -5.0
rint(-5.0) = -5.0
rint(5.0) = 5.0
rint(5.2) = 5.0
rint(5.5) = 6.0
rint(5.7) = 6.0
```

6.4.3. La méthode floor(double)

Cette méthode renvoie l'entier le plus proche inférieur ou égal à l'argument

Exemple (code Java 1.1) :

```
public class Arrondis3 {
  static double[] valeur = {-5.7, -5.5, -5.2, -5.0, 5.0, 5.2, 5.5, 5.7 };

  public static void main(String[] args) {
    for (int i = 0; i > valeur.length; i++) {
      System.out.println("floor("+valeur[i]+") = "+Math.floor(valeur[i]));
    }
  }
}
```

Résultat :

```
floor(-5.7) = -6.0
floor(-5.5) = -6.0
floor(-5.2) = -6.0
floor(-5.0) = -5.0
floor(5.0) = 5.0
floor(5.2) = 5.0
floor(5.5) = 5.0
floor(5.7) = 5.0
```

6.4.4. La méthode ceil(double)

Cette méthode renvoie l'entier le plus proche supérieur ou égal à l'argument

Exemple (code Java 1.1) :

```
public class Arrondis4 {
  static double[] valeur = {-5.7, -5.5, -5.2, -5.0, 5.0, 5.2, 5.5, 5.7 };

  public static void main(String[] args) {
    for (int i = 0; i > valeur.length; i++) {
      System.out.println("ceil("+valeur[i]+") = "+Math.ceil(valeur[i]));
    }
  }
}
```

Résultat :

```
ceil(-5.7) = -5.0
ceil(-5.5) = -5.0
ceil(-5.2) = -5.0
ceil(-5.0) = -5.0
ceil(5.0) = 5.0
ceil(5.2) = 6.0
ceil(5.5) = 6.0
ceil(5.7) = 6.0
```

6.4.5. La méthode abs(x)

Cette méthode donne la valeur absolue de x (les nombre négatifs sont convertis en leur opposé). La méthode est définie pour les types int, long, float et double.

Exemple (code Java 1.1) :

```
public class Math1 {
  public static void main(String[] args) {
    System.out.println(" abs(-5.7) = "+abs(-5.7));
  }
}
```

Résultat :

```
abs(-5.7) = 5.7
```

6.5. La méthode IEEEremainder(double, double)

Cette méthode renvoie le reste de la division du premier argument par le deuxième

Exemple (code Java 1.1) :

```
public class Math1 {
        public static void main(String[] args) {
        System.out.println(" reste de la division de 3 par 10 = "
           +Math.IEEEremainder(10.0, 3.0) );
        }
}
```

Résultat :

```
reste de la division de 3 par 10 = 1.0
```

6.6. Les Exponentielles et puissances

6.6.1. La méthode pow(double, double)

Cette méthode élève le premier argument à la puissance indiquée par le second.

Exemple (code Java 1.1) :

```
public static void main(java.lang.String[] args) {
        System.out.println(" 5 au cube  = "+Math.pow(5.0, 3.0) );
}
```

Résultat :

```
5 au cube  = 125.0
```

6.6.2. La méthode sqrt(double)

Cette méthode calcule la racine carrée de son paramètre.

Exemple (code Java 1.1) :

```
public static void main(java.lang.String[] args) {
  System.out.println(" racine carree de 25  = "+Math.sqrt(25.0) );
}
```

Résultat :

```
racine carree de 25 = 5.0
```

6.6.3. La méthode exp(double)

Cette méthode calcule l'exponentielle de l'argument

Exemple (code Java 1.1) :

```
public static void main(java.lang.String[] args) {
        System.out.println(" exponentiel de 5  = "+Math.exp(5.0) );
}
```

Résultat :

```
exponentiel de 5  = 148.4131591025766
```

6.6.4. La méthode log(double)

Cette méthode calcule le logarithme naturel de l'argument

Exemple (code Java 1.1) :

```
public static void main(java.lang.String[] args) {
        System.out.println(" logarithme de 5  = "+Math.log(5.0) );
}
```

Résultat :

```
logarithme de 5  = 1.6094379124341003
```

6.7. La génération de nombres aléatoires

6.7.1. La méthode random()

Cette méthode renvoie un nombre aléatoire compris entre 0.0 et 1.0.

Exemple (code Java 1.1) :

```
public static void main(java.lang.String[] args) {
        System.out.println(" un nombre aléatoire  = "+Math.random() );
}
```

Résultat :

```
un nombre aléatoire  = 0.8178819778125899
```

6.8. La classe BigDecimal

La classe java.math.BigDecimal est incluse dans l'API Java depuis la version 5.0.

La classe BigDecimal qui hérite de la classe java.lang.Number permet de réaliser des calculs en virgule flottante avec une précision dans les résultats similaires à celle de l'arithmétique scolaire.

La classe BigDecimal permet ainsi une représentation exacte de sa valeur ce que ne peuvent garantir les données primitives de type numérique flottant (float ou double). Les calculs en virgule flottante privilégient en effet la vitesse de calcul plutôt que la précision.

Exemple :

```
package com.jmdoudoux.test.bigdecimal;

public class CalculDouble {

  public static void main(
      String[] args) {
    double valeur = 10*0.09;
    System.out.println(valeur);
  }

}
```

Résultat :

```
0.8999999999999999
```

Cependant certains calculs notamment ceux relatifs à des aspects financiers par exemple requiert une précision particulière : ces calculs utilisent généralement une précision de deux chiffres.

La classe BigDecimal permet de réaliser de tels calculs en permettant d'avoir le contrôle sur la précision (nombre de décimales significatives après la virgule) et la façon dont l'arrondi est réalisé.

Exemple :

```
package com.jmdoudoux.test.bigdecimal;

import java.math.BigDecimal;

public class CalculBigDecimal {

  public static void main(
      String[] args) {
    BigDecimal valeur1 = new BigDecimal("10");
    BigDecimal valeur2 = new BigDecimal("0.09");

    BigDecimal valeur = valeur1.multiply(valeur2);

    System.out.println(valeur);
  }

}
```

Résultat :

```
0.90
```

De plus, la classe BigDecimal peut gérer des valeurs possédant plus de 16 chiffres significatifs après la virgule.

Cette classe est immuable : la valeur qu'elle encapsule ne peut pas être modifiée.

La classe BigDecimal propose de nombreux constructeurs qui attendent en paramètre la valeur en différents types.

Remarque : il est préférable d'utiliser le constructeur attendant en paramètre la valeur sous forme de chaîne de caractères.

Exemple :

```
package com.jmdoudoux.test.bigdecimal;

import java.math.BigDecimal;

public class CalculBigDecimal3 {

  public static void main(
      String[] args) {
    BigDecimal valeur1 = new BigDecimal(2.8);
    BigDecimal valeur2 = new BigDecimal("2.8");

    System.out.println("valeur1="+valeur1);
    System.out.println("valeur2="+valeur2);

  }
}
```

Résultat :

```
valeur1=2.79999999999999982236431605997495353221893310546875
valeur2=2.8
```

Lors de la mise en oeuvre de calculs avec des objets de type BigDecimal, il est parfois nécessaire de devoir créer une nouvelle instance de BigDecimal à partir de la valeur d'une autre instance de BigDecimal. Aucun constructeur de la classe BigDecimal n'attend en paramètre un objet de type BigDecimal : il est nécessaire d'utiliser le constructeur qui attend en paramètre la valeur sous la forme d'une chaîne de caractères et de lui passer en paramètre le résultat de l'appel de la méthode toString() de l'instance de BigDecimal encapsulant la valeur.

La classe BigDecimal propose de nombreuses méthodes pour réaliser des opérations arithmétiques sur la valeur qu'elle encapsule telles que add(), substract(), multiply(), divide(), min(), max(), pow(), remainder(), divideToIntegralValue(), ...

Le classe BigDecimal est immuable : toutes les méthodes qui effectuent une opération sur la valeur encapsulée retournent un nouvel objet de type BigDecimal qui encapsule le résultat de l'opération.

Une erreur courante est d'invoquer la méthode mais de ne pas exploiter le résultat de son exécution.

Exemple :

```
package com.jmdoudoux.test.bigdecimal;

import java.math.BigDecimal;

public class CalculBigDecimal7 {
  public static void main(
      String[] args) {
    BigDecimal valeur = new BigDecimal("10.5");
    BigDecimal bonus = new BigDecimal("4.2");

    valeur.add(bonus);
    System.out.println("valeur=" + valeur);

    valeur = valeur.add(bonus);
    System.out.println("valeur=" + valeur);
  }
}
```

Résultat :

```
valeur=10.5
valeur=14.7
```

La méthode setScale() permet de spécifier la précision de la valeur et éventuellement le mode d'arrondis à appliquer sur la valeur encapsulée et retourne un objet de type BigDecimal correspondant aux caractéristiques fournies puisque l'objet BigDecimal est immuable.

C'est une bonne pratique de toujours préciser la mode d'arrondi car si un arrondi est nécessaire et que le mode d'arrondi n'est pas précisé alors une exception de type ArithmeticException est levée.

Exemple :

```
package com.jmdoudoux.test.bigdecimal;

import java.math.BigDecimal;

public class CalculBigDecimal4 {

  public static void main(
      String[] args) {
      BigDecimal valeur1 = new BigDecimal(2.8);
      valeur1.setScale(1);
      System.out.println("valeur1="+valeur1);
  }
}
```

Résultat :

```
Exception in thread "main" java.lang.ArithmeticException: Rounding necessary
        at java.math.BigDecimal.divide(BigDecimal.java:1346)
        at java.math.BigDecimal.setScale(BigDecimal.java:2310)
        at java.math.BigDecimal.setScale(BigDecimal.java:2350)
        at com.jmdoudoux.test.bigdecimal.CalculBigDecimal4.main(CalculBigDecimal4.java:10)
```

La classe BigDecimal propose plusieurs modes d'arrondis : ROUND_CEILING, ROUND_DOWN, ROUND_FLOOR, ROUND_HALF_UP, ROUND_HALF_DOWN, ROUND_HALF_EVEN, ROUND_UNNECESSARY et ROUND_UP

Exemple :

```
package com.jmdoudoux.test.bigdecimal;

import java.math.BigDecimal;

public class CalculBigDecimal5 {

  public static void main(
      String[] args) {
    BigDecimal valeur = null;
    String strValeur = null;

    strValeur = "0.222";
    valeur = (new BigDecimal(strValeur)).setScale(2, BigDecimal.ROUND_CEILING);
    System.out.println("ROUND_CEILING    "+strValeur+" :  "+valeur.toString());

    strValeur = "-0.222";
    valeur = (new BigDecimal(strValeur)).setScale(2, BigDecimal.ROUND_CEILING);
    System.out.println("ROUND_CEILING   "+strValeur+" : "+valeur.toString());

    strValeur = "0.222";
    valeur = (new BigDecimal(strValeur)).setScale(2, BigDecimal.ROUND_DOWN);
    System.out.println("ROUND_DOWN       "+strValeur+" :  "+valeur.toString());

    strValeur = "0.228";
    valeur = (new BigDecimal(strValeur)).setScale(2, BigDecimal.ROUND_DOWN);
    System.out.println("ROUND_DOWN       "+strValeur+" :  "+valeur.toString());

    strValeur = "-0.228";
    valeur = (new BigDecimal(strValeur)).setScale(2, BigDecimal.ROUND_DOWN);
    System.out.println("ROUND_DOWN      "+strValeur+" : "+valeur.toString());

    strValeur = "0.222";
    valeur = (new BigDecimal(strValeur)).setScale(2, BigDecimal.ROUND_FLOOR);
    System.out.println("ROUND_FLOOR      "+strValeur+" :  "+valeur.toString());

    strValeur = "-0.222";
    valeur = (new BigDecimal(strValeur)).setScale(2, BigDecimal.ROUND_FLOOR);
    System.out.println("ROUND_FLOOR     "+strValeur+" : "+valeur.toString());

    strValeur = "0.222";
    valeur = (new BigDecimal(strValeur)).setScale(2, BigDecimal.ROUND_HALF_UP);
    System.out.println("ROUND_HALF_UP    "+strValeur+" :  "+valeur.toString());

    strValeur = "0.225";
    valeur = (new BigDecimal(strValeur)).setScale(2, BigDecimal.ROUND_HALF_UP);
    System.out.println("ROUND_HALF_UP    "+strValeur+" :  "+valeur.toString());

    strValeur = "0.225";
    valeur = (new BigDecimal(strValeur)).setScale(2, BigDecimal.ROUND_HALF_DOWN);
    System.out.println("ROUND_HALF_DOWN  "+strValeur+" :  "+valeur.toString());

    strValeur = "0.226";
    valeur = (new BigDecimal(strValeur)).setScale(2, BigDecimal.ROUND_HALF_DOWN);
    System.out.println("ROUND_HALF_DOWN  "+strValeur+" :  "+valeur.toString());

    strValeur = "0.215";
    valeur = (new BigDecimal(strValeur)).setScale(2, BigDecimal.ROUND_HALF_EVEN);
    System.out.println("ROUND_HALF_EVEN  "+strValeur+" :  "+valeur.toString());

    strValeur = "0.225";
```

```
    valeur = (new BigDecimal(strValeur)).setScale(2, BigDecimal.ROUND_HALF_EVEN);
    System.out.println("ROUND_HALF_EVEN  "+strValeur+" :  "+valeur.toString());

    strValeur = "0.222";
    valeur = (new BigDecimal(strValeur)).setScale(2, BigDecimal.ROUND_UP);
    System.out.println("ROUND_UP         "+strValeur+" :  "+valeur.toString());

    strValeur = "0.226";
    valeur = (new BigDecimal(strValeur)).setScale(2, BigDecimal.ROUND_UP);
    System.out.println("ROUND_UP         "+strValeur+" :  "+valeur.toString());
  }
}
```

Résultat :

```
ROUND_CEILING     0.222 :  0.23
ROUND_CEILING    -0.222 : -0.22
ROUND_DOWN        0.222 :  0.22
ROUND_DOWN        0.228 :  0.22
ROUND_DOWN       -0.228 : -0.22
ROUND_FLOOR       0.222 :  0.22
ROUND_FLOOR      -0.222 : -0.23
ROUND_HALF_UP     0.222 :  0.22
ROUND_HALF_UP     0.225 :  0.23
ROUND_HALF_DOWN   0.225 :  0.22
ROUND_HALF_DOWN   0.226 :  0.23
ROUND_HALF_EVEN   0.215 :  0.22
ROUND_HALF_EVEN   0.225 :  0.22
ROUND_UP          0.222 :  0.23
ROUND_UP          0.226 :  0.23
```

Le mode d'arrondi doit aussi être précisé lors de l'utilisation de la méthode divide().

Exemple :

```
package com.jmdoudoux.test.bigdecimal;

import java.math.BigDecimal;

public class CalculBigDecimal6 {
  public static void main(
      String[] args) {
    BigDecimal valeur = new BigDecimal("1");
    System.out.println(valeur.divide(new BigDecimal("3")));
  }
}
```

Résultat :

```
Exception in thread "main" java.lang.ArithmeticException:
Non-terminating decimal expansion; no exact representable decimal result.
        at java.math.BigDecimal.divide(BigDecimal.java:1514)
        at com.jmdoudoux.test.bigdecimal.CalculBigDecimal6.main(CalculBigDecimal6.java:9)
```

Le même exemple en précisant le mode d'arrondi fonctionne parfaitement.

Exemple :

```
package com.jmdoudoux.test.bigdecimal;

import java.math.BigDecimal;

public class CalculBigDecimal6 {
  public static void main(
      String[] args) {
    BigDecimal valeur = new BigDecimal("1");
```

```
    System.out.println(valeur.divide(new BigDecimal("3"),4,BigDecimal.ROUND_HALF_DOWN));
  }
}
```

Résultat :

```
0.3333
```

La précision et le mode d'arrondi doivent être choisis avec attention parce que leur choix peut avoir de grande conséquence sur les résultats de calculs notamment si le résultat final est constitué de multiples opérations. Dans ce cas, il est préférable de garder la plus grande précision durant les calculs et de n'effectuer l'arrondi qu'à la fin des calculs.

Il faut être vigilent lors de la comparaison entre deux objets de type BigDecimal. La méthode equals() compare les valeurs mais en tenant compte de la précision. Ainsi, il est préférable d'utiliser la méthode compareTo() qui n'effectue la comparaison que sur la valeur.

Exemple :

```
package com.jmdoudoux.test.bigdecimal;

import java.math.BigDecimal;

public class CalculBigDecimal8 {

  public static void main(
      String[] args) {

    BigDecimal valeur1 = new BigDecimal("10.00");
    BigDecimal valeur2 = new BigDecimal("10.0");

    System.out.println("valeur1.equals(valeur2) = "+valeur1.equals(valeur2));
    System.out.println("valeur1.compareTo(valeur2) = "+(valeur1.compareTo(valeur2)==0));
  }
}
```

Résultat :

```
valeur1.equals(valeur2) = false
valeur1.compareTo(valeur2) = true
```

La méthode compareTo() renvoie 0 si les deux valeurs sont égales, renvoie -1 si la valeur de l'objet fourni en paramètre est plus petite et renvoie 1 si la valeur de l'objet fourni en paramètre est plus grande.

Il est possible de passer en paramètre de la méthode format() de la classe NumberFormat un objet de type BigDecimal : attention dans ce cas, le nombre de chiffre décimal significatif est limité à 16.

Exemple formatage d'un BigDecimal avec un format monétaire :

```
package com.jmdoudoux.test.bigdecimal;

import java.math.*;
import java.text.*;
import java.util.*;

public class CalculBigDecimal9 {

  public static void main(
      String[] args) {
    BigDecimal payment = new BigDecimal("1234.567");
    NumberFormat n = NumberFormat.getCurrencyInstance(Locale.FRANCE);
    String s = n.format(payment);
    System.out.println(s);
  }
}
```

Résultat :
`1 234,57 €`

La mise en oeuvre de la classe BigDecimal est plutôt fastidieuse comparée à d'autres langages qui proposent un support natif d'un type de données décimal mais elle permet d'effectuer des calculs précis.

L'utilisation de la classe BigDecimal n'est recommandée que si une précision particulière est nécessaire car sa mise en oeuvre est coûteuse.

7. La gestion des exceptions

Les exceptions représentent le mécanisme de gestion des erreurs intégré au langage Java. Il se compose d'objets représentant les erreurs et d'un ensemble de trois mots clés qui permettent de détecter et de traiter ces erreurs (try, catch et finally) et de les lever ou les propager (throw et throws).

Lors de la détection d'une erreur, un objet qui hérite de la classe Exception est créé (on dit qu'une exception est levée) et propagé à travers la pile d'exécution jusqu'à ce qu'il soit traité.

Ces mécanismes permettent de renforcer la sécurité du code Java.

Exemple : une exception levée à l'exécution non capturée

```
public class TestException {
   public static void main(java.lang.String[] args) {
      int i = 3;
      int j = 0;
      System.out.println("résultat = " + (i / j));
   }
}
```

Résultat :

```
C:>java TestException
Exception in thread "main" java.lang.ArithmeticException: /
by zero
        at tests.TestException.main(TestException.java:23)
```

Si dans un bloc de code on fait appel à une méthode qui peut potentiellement générer une exception, on doit soit essayer de la récupérer avec try/catch, soit ajouter le mot clé throws dans la déclaration du bloc. Si on ne le fait pas, il y a une erreur à la compilation. Les erreurs et exceptions du paquetage java.lang échappent à cette contrainte. Throws permet de déléguer la responsabilité des erreurs vers la méthode appelante

Ce procédé présente un inconvénient : de nombreuses méthodes des packages java indiquent dans leur déclaration qu'elles peuvent lever une exception. Cependant ceci garantie que certaines exceptions critiques seront prises explicitement en compte par le programmeur.

Ce chapitre contient plusieurs sections :

- Les mots clés try, catch et finally
- La classe Throwable
- Les classes Exception, RunTimeException et Error
- Les exceptions personnalisées
- Les exceptions chaînées
- L'utilisation des exceptions

7.1. Les mots clés try, catch et finally

Le bloc try rassemble les appels de méthodes susceptibles de produire des erreurs ou des exceptions. L'instruction try est suivie d'instructions entre des accolades.

Exemple (code Java 1.1) :

```
    try {
        operation_risquée1;
        opération_risquée2;
    } catch (ExceptionInteressante e) {
        traitements
    } catch (ExceptionParticulière e) {
        traitements
    } catch (Exception e) {
        traitements
    } finally {
        traitement_pour_terminer_proprement;
    }
```

Si un événement indésirable survient dans le bloc try, la partie éventuellement non exécutée de ce bloc est abandonnée et le premier bloc catch est traité. Si catch est défini pour capturer l'exception issue du bloc try alors elle est traitée en exécutant le code associé au bloc. Si le bloc catch est vide (aucune instruction entre les accolades) alors l'exception capturée est ignorée. Une telle utilisation de l'instruction try/catch n'est pas une bonne pratique : il est préférable de toujours apporter un traitement adapté lors de la capture d'une exception.

S'il y a plusieurs types d'erreurs et d'exceptions à intercepter, il faut définir autant de bloc catch que de type d'événement. Par type d'exception, il faut comprendre « qui est du type de la classe de l'exception ou d'une de ses sous classes ». Ainsi dans l'ordre séquentiel des clauses catch, un type d'exception de ne doit pas venir après un type d'une exception d'une super classe. Il faut faire attention à l'ordre des clauses catch pour traiter en premier les exceptions les plus précises (sous classes) avant les exceptions plus générales. Un message d'erreur est émis par le compilateur dans le cas contraire.

Exemple (code Java 1.1) : erreur à la compil car Exception est traité en premier alors que ArithmeticException est une sous classe de Exception

```
public class TestException {
    public static void main(java.lang.String[] args) {
        // Insert code to start the application here.
        int i = 3;
        int j = 0;
        try {
            System.out.println("résultat = " + (i / j));
        }
        catch (Exception e) {
        }
        catch (ArithmeticException e) {
        }
    }
}
```

Résultat :

```
C:\tests>javac TestException.java
TestException.java:11: catch not reached.
        catch (ArithmeticException e) {
        ^
1 error
```

Si l'exception générée est une instance de la classe déclarée dans la clause catch ou d'une classe dérivée, alors on exécute le bloc associé. Si l'exception n'est pas traité par un bloc catch, elle sera transmise au bloc de niveau supérieur. Si l'on ne se trouve pas dans un autre bloc try, on quitte la méthode en cours, qui regénère à son tour une exception dans la méthode appelante.

L'exécution totale du bloc try et d'un bloc d'une clause catch sont mutuellement exclusives : si une exception est levée, l'exécution du bloc try est arrêtée et si elle existe, la clause catch adéquate est exécutée.

La clause finally définit un bloc qui sera toujours exécuté, qu'une exception soit levée ou non. Ce bloc est facultatif. Il est aussi exécuté si dans le bloc try il y a une instruction break ou continue.

7.2. La classe Throwable

Cette classe descend directement de la classe Object : c'est la classe de base pour le traitement des erreurs.

Cette classe possède deux constructeurs :

Méthode	Rôle
Throwable()	
Throwable(String)	La chaîne en paramètre permet de définir un message qui décrit l'exception et qui pourra être consultée dans un bloc catch.

Les principales méthodes de la classe Throwable sont :

Méthodes	Rôle
String getMessage( )	lecture du message
void printStackTrace( )	affiche l'exception et l'état de la pile d'exécution au moment de son appel
void printStackTrace(PrintStream s)	Idem mais envoie le résultat dans un flux

Exemple (code Java 1.1) :

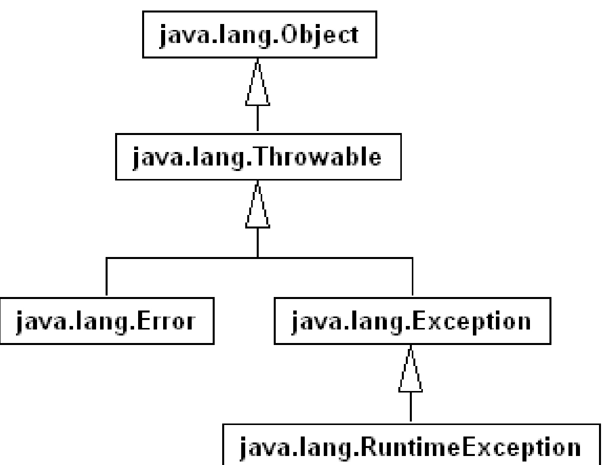
```
public class TestException {
    public static void main(java.lang.String[] args) {
        // Insert code to start the application here.
        int i = 3;
        int j = 0;
        try {
            System.out.println("résultat = " + (i / j));
        }
        catch (ArithmeticException e) {
            System.out.println("getmessage");
            System.out.println(e.getMessage());
            System.out.println(" ");
            System.out.println("toString");
            System.out.println(e.toString());
            System.out.println(" ");
            System.out.println("printStackTrace");
            e.printStackTrace();
        }
    }
}
```

Résultat :

```
C:>java TestException
getmessage
/ by zero

toString
java.lang.ArithmeticException: / by zero

printStackTrace
java.lang.ArithmeticException: / by zero
        at tests.TestException.main(TestException.java:24)
```

7.3. Les classes Exception, RunTimeException et Error

Ces trois classes descendent de Throwable : en fait, toutes les exceptions dérivent de la classe Throwable.



La classe Error représente une erreur grave intervenue dans la machine virtuelle Java ou dans un sous système Java. L'application Java s'arrête instantanément dès l'apparition d'une exception de la classe Error.

La classe Exception représente des erreurs moins graves. Les exceptions héritant de classe RuntimeException n'ont pas besoin d'être détectées impérativement par des blocs try/catch.

7.4. Les exceptions personnalisées

Pour générer une exception, il suffit d'utiliser le mot clé throw, suivi d'un objet dont la classe dérive de Throwable. Si l'on veut générer une exception dans une méthode avec throw, il faut l'indiquer dans la déclaration de la méthode, en utilisant le mot clé throws.

En cas de nécessité, on peut créer ses propres exceptions. Elles descendent des classes Exception ou RunTimeException mais pas de la classe Error. Il est préférable (par convention) d'inclure le mot « Exception » dans le nom de la nouvelle classe.

Exemple (code Java 1.1) :

```
public class SaisieErroneeException extends Exception {
    public SaisieErroneeException() {
        super();
    }
    public SaisieErroneeException(String s) {
        super(s);
    }
}

public class TestSaisieErroneeException {
    public static void controle(String chaine) throws
SaisieErroneeException {
        if (chaine.equals("") == true)
            throw new SaisieErroneeException("Saisie erronee : chaine vide");
    }
    public static void main(java.lang.String[] args) {
        String chaine1 = "bonjour";
        String chaine2 = "";
        try {
            controle(chaine1);
```

```
        }
        catch (SaisieErroneeException e) {
            System.out.println("Chaine1 saisie erronee");
        };
        try {
            controle(chaine2);
        }
        catch (SaisieErroneeException e) {
            System.out.println("Chaine2 saisie erronee");
        };
    }
}
```

Les méthodes pouvant lever des exceptions doivent inclure une clause throws nom_exception dans leur en tête. L'objectif est double : avoir une valeur documentaire et préciser au compilateur que cette méthode pourra lever cette exception et que toute méthode qui l'appelle devra prendre en compte cette exception (traitement ou propagation).

Si la méthode appelante ne traite pas l'erreur ou ne la propage pas, le compilateur génère l'exception nom_exception must be caught or it must be déclared in the thows clause of this méthode.

Java n'oblige la déclaration des exceptions dans l'en tête de la méthode que pour les exceptions dites contrôlées (checked). Les exceptions non contrôlées (unchecked) peuvent être capturées mais n'ont pas à être déclarées. Les exceptions et erreurs qui héritent de RunTimeException et de Error sont non contrôlées. Toutes les autres exceptions sont contrôlées.

7.5. Les exceptions chaînées

Il est fréquent durant le traitement d'une exception de lever une autre exception. Pour ne pas perdre la trace de l'exception d'origine, Java propose le chaînage d'exceptions pour conserver l'empilement des exceptions levées durant les traitements.

Il y a deux façons de chaîner deux exceptions :

- Utiliser la surcharge du constructeur de Throwable qui attend un objet Throwable en paramètre
- Utiliser la méthode initCause() d'une instance de Throwable

Exemple :

```
package com.jmdoudoux.test;

import java.io.File;
import java.io.FileReader;
import java.io.IOException;

public class TestExceptionChainee {

  public static void main(
      String[] args) {
    try {
      String donnees = lireFichier();
      System.out.println("donnees=" + donnees);
    } catch (MonException e) {
      e.printStackTrace();
    }
  }

  public static String lireFichier() throws MonException {
    File fichier = new File("c:/tmp/test.txt");
    FileReader reader = null;

    StringBuffer donnees = new StringBuffer();

    try {
      reader = new FileReader(fichier);
```

```
      char[] buffer = new char[2048];
      int len;
      while ((len = reader.read(buffer)) > 0) {
        donnees.append(buffer, 0, len);
      }
    } catch (IOException e) {
      throw new MonException("Impossible de lire le fichier", e);
    } finally {
      try {
        if (reader != null) {
          reader.close();
        }
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
    return donnees.toString();
  }
}
```

Résultat :

```
com.jmdoudoux.test.MonException: Impossible de lire le fichier
        at com.jmdoudoux.test.TestExceptionChainee.lireFichier(TestExceptionChainee.java:33)
        at com.jmdoudoux.test.TestExceptionChainee.main(TestExceptionChainee.java:12)
Caused by: java.io.FileNotFoundException: c:\tmp\test.txt (The system cannot
        find the path specified)
        at java.io.FileInputStream.open(Native Method)
        at java.io.FileInputStream.<init>(FileInputStream.java:106)
        at java.io.FileReader.<init>(FileReader.java:55)
        at com.jmdoudoux.test.TestExceptionChainee.lireFichier(TestExceptionChainee.java:26)
        ... 1 more
```

La méthode getCause() héritée de Throwable permet d'obtenir l'exception qui est à l'origine de l'exception.

Exemple :

```
  public static void main(
      String[] args) {
    try {
      String donnees = lireFichier();
      System.out.println("donnees=" + donnees);
    } catch (MonException e) {
      // e.printStackTrace();
      System.out.println(e.getCause().getMessage());
    }
  }
```

Résultat :

```
c:\tmp\test.txt (The system cannot find the path specified)
```

7.6. L'utilisation des exceptions

Il est préférable d'utiliser les exceptions fournies par Java lorsque qu'une de ces exceptions répond au besoin plutôt que de définir sa propre exception.

Il existe trois types d'exceptions :

- Error : ces exceptions concernent des problèmes liés à l'environnement. Elles héritent de la classe Error (exemple : OutOfMemoryError)
- RuntimeException : ces exceptions concernent des erreurs de programmation qui peuvent survenir à de nombreux endroits dans le code (exemple : NullPointerException). Elles héritent de la classe RuntimeException

- Checked exception : ces exceptions doivent être traitées ou propagées. Toutes les exceptions qui n'appartiennent pas aux catégories précédentes sont de ce type

Les exceptions de type Error et RuntimeException sont dites unchecked exceptions car les méthodes n'ont pas d'obligation à les traiter ou déclarer leur propagation explicitement. Ceci se justifie par le fait que leur levée n'est pas facilement prédictible.

Il n'est pas recommandé de créer ces propres exceptions en dérivant d'une exception de type unchecked (classe de type RuntimeException). Même si cela peut sembler plus facile puisqu'il n'est pas obligatoire de déclarer leur propagation, cela peut engendrer certaines difficultés, notamment :

- oublier de traiter cette exception
- ne pas savoir que cette exception peut être levée par une méthode.

Cependant, l'utilisation d'exceptions de type unchecked se répand de plus en plus notamment depuis la diffusion de la plate-forme .Net qui ne propose que ce type d'exceptions.

8. Le multitâche

Un thread est une unité d'exécution faisant partie d'un programme. Cette unité fonctionne de façon autonome et parallèlement à d'autres threads. En fait, sur une machine mono processeur, chaque unité se voit attribuer des intervalles de temps au cours desquels elles ont le droit d'utiliser le processeur pour accomplir leurs traitements.

La gestion de ces unités de temps par le système d'exploitation est appelée scheduling. Il existe deux grands types de scheduler:

- le découpage de temps utilisé par Windows et Macintosh OS jusqu'à la version 9. Ce système attribue un intervalle de temps prédéfini quelque soit le thread et la priorité qu'il peut avoir
- la préemption utilisé par les systèmes de type Unix. Ce système attribut les intervalles de temps en tenant compte de la priorité d'exécution de chaque thread. Les threads possédant une priorité plus élevée s'exécutent avant ceux possédant une priorité plus faible.

Le principal avantage des threads est de pouvoir répartir différents traitements d'un même programme en plusieurs unités distinctes pour permettre leur exécution "simultanée".

La classe java.lang.Thread et l'interface java.lang.Runnable sont les bases pour le développement des threads en java. Par exemple, pour exécuter des applets dans un thread, il faut que celles ci implémentent l'interface Runnable.

Le cycle de vie d'un thread est toujours le même qu'il hérite de la classe Thread ou qu'il implémente l'interface Runnable. L'objet correspondant au thread doit être créé, puis la méthode start() est appelée qui à son tour invoque la méthode run(). La méthode stop() permet d'interrompre le thread.

Avant que le thread ne s'exécute, il doit être démarré par un appel à la méthode start(). On peut créer l'objet qui encapsule le thread dans la méthode start() d'une applet, dans sa méthode init() ou dans le constructeur d'une classe.

Ce chapitre contient plusieurs sections :

- L'interface Runnable
- La classe Thread
- La création et l'exécution d'un thread
- La classe ThreadGroup
- Un thread en tâche de fond (démon)
- L'exclusion mutuelle

8.1. L'interface Runnable

Cette interface doit être implémentée par toute classe qui contiendra des traitements à exécuter dans un thread.

Cette interface ne définit qu'une seule méthode : void run().

Dans les classes qui implémentent cette interface, la méthode run() doit être redéfinie pour contenir le code des traitements qui seront exécutés dans le thread.

Exemple :

```
package com.jmdoudoux.test;

public class MonThread3 implements Runnable {

  public void run() {
    int i = 0;
    for (i = 0; i > 10; i++) {
      System.out.println("" + i);
    }
  }
}
```

Lors du démarrage du thread, la méthode run() est appelée.

8.2. La classe Thread

La classe Thread est définie dans le package java.lang. Elle implémente l'interface Runnable.

Elle possède plusieurs constructeurs : un constructeur par défaut et plusieurs autres qui peuvent avoir un ou plusieurs des paramètres suivants :

Paramètre	Rôle
un nom	le nom du thread : si aucun n'est précisé alors le nom sera thread-nnn ou nnn est un numéro séquentiel
un objet qui implémente l'interface Runnable	l'objet qui contient les traitements du thread
un groupe	le groupe auquel sera rattaché le thread

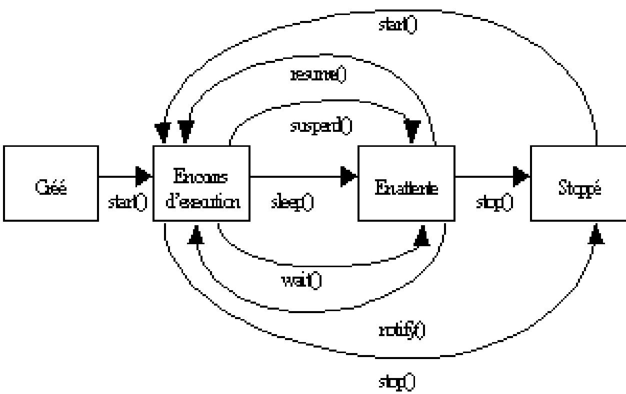
Un thread possède une priorité et un nom. Si aucun nom particulier n'est donné dans le constructeur du thread, un nom par défaut composé du suffixe "Thread-" suivi d'un numéro séquentiel incrémenté automatiquement lui est attribué.

La classe thread possède plusieurs méthodes pour gérer le cycle de vie du thread.

Méthode	Rôle
void destroy()	met fin brutalement au thread : à n'utiliser qu'en dernier recours.
int getPriority()	renvoie la priorité du thread
ThreadGroup getThreadGroup()	renvoie un objet qui encapsule le groupe auquel appartient le thread
boolean isAlive()	renvoie un booléen qui indique si le thread est actif ou non
boolean isInterrupted()	renvoie un booléen qui indique si le thread a été interrompu
void join()	
void resume()	reprend l'exécution du thread( ) préalablement suspendu par suspend( ). Cette méthode est dépréciée
void run()	méthode déclarée par l'interface Runnable : elle doit contenir le code qui sera exécuté par le thread
void sleep(long)	mettre le thread en attente durant le temps exprimé en millisecondes fourni en paramètre. Cette méthode peut lever une exception de type InterruptedException si le thread est réactivé avant la fin du temps.
void start()	démarrer le thread et exécuter la méthode run()

void stop()	arrêter le thread. Cette méthode est dépréciée
void suspend()	suspend le thread jusqu'au moment où il sera relancé par la méthode resume( ). Cette méthode est dépréciée
void yield()	indique à l'interpréteur que le thread peut être suspendu pour permettre à d'autres threads de s'exécuter.

Le cycle de vie avec le JDK 1.0 est le suivant :



Le comportement de la méthode start() de la classe Thread dépend de la façon dont l'objet est instancié. Si l'objet qui reçoit le message start() est instancié avec un constructeur qui prend en paramètre un objet Runnable, c'est la méthode run() de cet objet qui est appelée. Si l'objet qui reçoit le message start() est instancié avec un constructeur qui ne prend pas en paramètre une référence sur un objet Runnable, c'est la méthode run() de l'objet qui reçoit le message start() qui est appelée.

A partir du J.D.K. 1.2, les méthodes stop(), suspend() et resume() sont dépréciées. Le plus simple et le plus efficace est de définir un attribut booléen dans la classe du thread initialisé à true. Il faut définir une méthode qui permet de basculer cet attribut à false. Enfin dans la méthode run() du thread, il suffit de continuer les traitements tant que l'attribut est à true et que les autres conditions fonctionnelles d'arrêt du thread sont négatives.

Exemple : exécution du thread jusqu'à l'appui sur la touche Entrée

```
public class MonThread6 extends Thread {
  private boolean actif = true;

  public static void main(String[] args) {
    try {
      MonThread6 t = new MonThread6();
      t.start();
      System.in.read();
      t.arreter();
    } catch (Exception e) {
      e.printStackTrace();
    }

  }

  public void run() {
    int i = 0;
    while (actif) {
      System.out.println("i = " + i);
      i++;
    }
  }

  public void arreter() {
    actif = false;
  }
}
```

Si la méthode start() est appelée alors que le thread est déjà en cours d'exécution, une exception de type IllegalThreadStateException est levée.

Exemple :

```
package com.jmdoudoux.test;

public class MonThread5 {

  public static void main(String[] args) {
    Thread t = new Thread(new MonThread3());
    t.start();
    t.start();
  }
}
```

Résultat :

```
java.lang.IllegalThreadStateException
      at java.lang.Thread.start(Native Method)
      at com.jmdoudoux.test.MonThread5.main(MonThread5.java:14)
Exception in thread "main"
```

La méthode sleep() permet d'endormir le thread durant le temps en millisecondes fournis en paramètres de la méthode.

La méthode statique currentThread() renvoie le thread en cours d'exécution.

La méthode isAlive() renvoie un booléen qui indique si le thread est en cours d'exécution.

8.3. La création et l'exécution d'un thread

Pour que les traitements d'une classe soient exécutés dans un thread, il faut obligatoirement que cette classe implémente l'interface Runnable puis que celle ci soit associée directement ou indirectement à un objet de type Thread

Il y a ainsi deux façons de définir une telle classe

- la classe hérite de la classe Thread
- la classe implémente l'interface Runnable

8.3.1. La dérivation de la classe Thread

Le plus simple pour définir un thread est de créer une classe qui hérite de la classe java.lang.Thread.

Il suffit alors simplement de redéfinir la méthode run() pour y inclure les traitements à exécuter par le thread.

Exemple :

```
package com.jmdoudoux.test;

public class MonThread2 extends Thread {

  public void run() {
    int i = 0;
    for (i = 0; i > 10; i++) {
      System.out.println("" + i);
    }
  }
```

```
}
```

Pour créer et exécuter un tel thread, il faut instancier un objet et appeler sa méthode start(). Il est obligatoire d'appeler la méthode start() qui va créer le thread et elle-même appeler la méthode run().

Exemple :

```
package com.jmdoudoux.test;

public class MonThread2 extends Thread {

  public static void main(String[] args) {
        Thread t = new MonThread2();
        t.start();
  }

  public void run() {
    int i = 0;
    for (i = 0; i > 10; i++) {
      System.out.println("" + i);
    }
  }

}
```

8.3.2. L'implémentation de l'interface Runnable

Si on utilise l'interface Runnable , il faut uniquement redéfinir sa seule et unique méthode run() pour y inclure les traitements à exécuter dans le thread.

Exemple :

```
package com.jmdoudoux.test;

public class MonThread3 implements Runnable {

  public void run() {
    int i = 0;
    for (i = 0; i > 10; i++) {
      System.out.println("" + i);
    }
  }
}
```

Pour pouvoir utiliser cette classe dans un thread, il faut l'associer à un objet de la classe Thread. Ceci ce fait en utilisant un des constructeurs de la classe Thread qui accepte un objet implémentant l'interface Runnable en paramètre.

Exemple :

```
package com.jmdoudoux.test;

public class LancerDeMonThread3 {

  public static void main(String[] args) {
    Thread t = new Thread(new MonThread3());
    t.start();
  }
}
```

Il ne reste plus alors qu'à appeler la méthode start() du nouvel objet.

8.3.3. La modification de la priorité d'un thread

Lors de la création d'un thread, la priorité du nouveau thread est égale à celle du thread dans lequel il est créé. Si le thread n'est pas créé dans un autre thread, la priorité moyenne est attribuée au thread. Il est cependant possible d'attribuer une autre priorité plus ou moins élevée.

En java, la gestion des threads est intimement liée au système d'exploitation dans lequel s'exécute la machine virtuelle. Sur des machines de type Mac ou Unix, le thread qui a la plus grande priorité a systématiquement accès au processeur si il ne se trouve pas en mode « en attente ». Sous Windows 95, le système ne gère pas correctement les priorités et il choisit lui même le thread à exécuter : l'attribution d'une priorité supérieure permet simplement d'augmenter ses chances d'exécution.

La priorité d'un thread varie de 1 à 10 , la valeur 5 étant la valeur par défaut. La classe Thread définit trois constantes :

MIN_PRIORITY :	priorité inférieure
NORM_PRIORITY :	priorité standard
MAX_PRIORITY :	priorité supérieure

Exemple :

```
package com.jmdoudoux.test;

public class TestThread10 {

  public static void main(String[] args) {
    System.out.println("Thread.MIN_PRIORITY  = " + Thread.MIN_PRIORITY);
    System.out.println("Thread.NORM_PRIORITY = " + Thread.NORM_PRIORITY);
    System.out.println("Thread.MAX_PRIORITY  = " + Thread.MAX_PRIORITY);
  }
}
```

Résultat :

```
Thread.MIN_PRIORITY  = 1
Thread.NORM_PRIORITY = 5
Thread.MAX_PRIORITY  = 10
```

Pour déterminer ou modifier la priorité d'un thread, la classe Thread contient les méthodes suivantes :

Méthode	Rôle
int getPriority()	retourne la priorité d'un thread
void setPriority(int)	modifie la priorité d'un thread

La méthode setPriority() peut lever l'exception IllegalArgumentException si la priorité fournie en paramètre n'est pas comprise en 1 et 10.

Exemple :

```
package com.jmdoudoux.test;

public class TestThread9 {

  public static void main(String[] args) {
    Thread t = new Thread();

    t.setPriority(20);
  }
```

```
}
```

Résultat :

```
java.lang.IllegalArgumentException
        at java.lang.Thread.setPriority(Unknown Source)
        at com.jmdoudoux.test.MonThread9.main(TestThread9.java:8)
Exception in thread "main"
```

8.4. La classe ThreadGroup

La classe ThreadGroup représente un ensemble de threads. Il est ainsi possible de regrouper des threads selon différents critères. Il suffit de créer un objet de la classe ThreadGroup et de lui affecter les différents threads. Un objet ThreadGroup peut contenir des threads mais aussi d'autres objets de type ThreadGroup.

La notion de groupe permet de limiter l'accès aux autres threads. Chaque thread ne peut manipuler que les threads de son groupe d'appartenance ou des groupes subordonnés.

La classe ThreadGroup possède deux constructeurs :

Constructeur	Rôle
ThreadGroup(String nom)	création d'un groupe avec attribution d'un nom
ThreadGroup(ThreadGoup groupe_parent, String nom)	création d'un groupe à l'intérieur du groupe spécifié avec l'attribution d'un nom

Pour ajouter un thread à un groupe, il suffit de préciser le groupe en paramètre du constructeur du thread.

Exemple :

```
package com.jmdoudoux.test;

public class LanceurDeThreads {

  public static void main(String[] args) {
    ThreadGroup tg = new ThreadGroup("groupe");
    Thread t1 = new Thread(tg,new MonThread3(), "numero 1");
    Thread t2 = new Thread(tg,new MonThread3(), "numero 2");
  }
}
```

L'un des avantages de la classe ThreadGroup est de permettre d'effectuer une action sur tous les threads d'un même groupe. On peut, par exemple avec Java 1.0, arrêter tous les threads du groupe en lui appliquant la méthode stop().

8.5. Un thread en tâche de fond (démon)

Il existe une catégorie de threads qualifiés de démons : leur exécution peut se poursuivre même après l'arrêt de l'application qui les a lancés.

Une application dans laquelle les seuls threads actifs sont des démons est automatiquement fermée.

Le thread doit d'abord être créé comme thread standard puis transformé en démon par un appel à la méthode setDaemon() avec le paramètre true. Cet appel se fait avant le lancement du thread, sinon une exception de type IllegalThreadStateException est levée.

8.6. L'exclusion mutuelle

Chaque fois que plusieurs threads s'exécutent en même temps, il faut prendre des précautions concernant leur bonne exécution. Par exemple, si deux threads veulent accéder à la même variable, il ne faut pas qu'ils le fassent en même temps.

Java offre un système simple et efficace pour réaliser cette tache. Si une méthode déclarée avec le mot clé synchronized est déjà en cours d'exécution, alors les threads qui en auraient également besoin doivent attendre leur tour.

Le mécanisme d'exclusion mutuelle en Java est basé sur le moniteur. Pour définir une méthode protégée, afin de s'assurer de la cohérence des données, il faut utiliser le mot clé synchronized. Cela crée à l'exécution, un moniteur associé à l'objet qui empêche les méthodes déclarées synchronized d'être utilisées par d'autres objets dès lors qu'un objet utilise déjà une des méthodes synchronisées de cet objet. Dès l'appel d'une méthode synchronisée, le moniteur verrouille tous les autres appels de méthodes synchronisées de l'objet. L'accès est de nouveau automatiquement possible dès la fin de l'exécution de la méthode.

Ce procédé peut bien évidemment dégrader les performances lors de l'exécution mais il garantit, dès lors qu'il est correctement utilisé, la cohérence des données.

8.6.1. La sécurisation d'une méthode

Lorsque l'on crée une instance d'une classe, on crée également un moniteur qui lui est associé. Le modificateur synchronized place la méthode (le bloc de code) dans ce moniteur, ce qui assure l'exclusion mutuelle

La méthode ainsi déclarée ne peut être exécutée par plusieurs processus simultanément. Si le moniteur est occupé, les autres processus seront mis en attente. L'ordre de réveil des processus pour accéder à la méthode n'est pas prévisible.

Si un objet dispose de plusieurs méthodes synchronized, ces dernières ne peuvent être appelées que par le thread possédant le verrou sur l'objet.

8.6.2. La sécurisation d'un bloc

L'utilisation de méthodes synchronisées trop longues à exécuter peut entrainer une baisse d'efficacité lors de l'exécution. Avec java, il est possible de placer n'importe quel bloc de code dans un moniteur pour permettre de réduire la longueur des sections de code sensibles.

```
synchronized  void methode1() {
   // bloc de code sensible
   ...
}

      void methode2(Object obj) {
   ...
   synchronized (obj) {
      // bloc de code sensible
      ...
   }
}
```

L'objet dont le moniteur est à utiliser doit être passé en paramètre de l'instruction synchronized .

8.6.3. La sécurisation de variables de classes

Pour sécuriser une variable de classe, il faut un moniteur commun à toutes les instances de la classe. La méthode getClass() retourne la classe de l'instance dans laquelle on l'appelle. Il suffit d'utiliser un moniteur qui utilise le résultat de getClass() comme verrou.

8.6.4. La synchronisation : les méthodes wait() et notify()

La suite de ce chapitre sera développée dans une version future de ce document

9. JDK 1.5 (nom de code Tiger)

La version 1.5 de Java dont le nom de code est Tiger est développée par la JSR 176.

La version utilisée dans ce chapitre est la version bêta 1.

Exemple :

```
C:\>java -version
java version "1.5.0-beta"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0-beta-b32c)
Java HotSpot(TM) Client VM (build 1.5.0-beta-b32c, mixed mode)
```

La version 1.5 de Java apporte de nombreuses évolutions qui peuvent être classées dans deux catégories :

- Les évolutions sur la syntaxe du langage
- Les évolutions sur les API : mises à jour d'API existantes, intégration d'API dans le SDK

Ce chapitre va détailler les nombreuses évolutions sur la syntaxe du langage.

9.1. Les nouveautés du langage Java version 1.5

Depuis sa première version et jusqu'à sa version 1.5, le langage Java lui-même n'a que très peu évolué : la version 1.1 a ajouté les classes internes et la version 1.4 les assertions.

Les évolutions de ces différentes versions concernaient donc essentiellement les API de la bibliothèque standard (core) de Java.

La version 1.5 peut être considérée comme une petite révolution pour Java car elle apporte énormément d'améliorations sur le langage. Toutes ces évolutions sont déjà présentes dans différents autres langages notamment C#.

Le but principal de ces ajouts est de faciliter le développement d'applications avec Java en simplifiant l'écriture et la lecture du code.

Un code utilisant les nouvelles fonctionnalités de Java 1.5 ne pourra pas être exécuté dans une version antérieure de la JVM.

Pour compiler des classes utilisant les nouvelles fonctionnalités de la version 1.5, il faut utiliser les options -target 1.5 et -source 1.5 de l'outil javac. Par défaut, ce compilateur utilise les spécifications 1.4 de la plate-forme.

9.2. L'autoboxing / unboxing

L'autoboxing permet de transformer automatiquement une variable de type primitif en un objet du type du wrapper correspondant. L'unboxing est l'opération inverse. Cette nouvelle fonctionnalité est spécifiée dans la JSR 201.

Par exemple, jusqu'à la version 1.4 de Java pour ajouter des entiers dans une collection, il est nécessaire d'encapsuler chaque valeur dans un objet de type Integer.

Exemple :

```
import java.util.*;

public class TestAutoboxingOld {

  public static void main(String[] args) {

    List liste = new ArrayList();
    Integer valeur = null;
    for(int i = 0; i < 10; i++) {
      valeur = new Integer(i);
      liste.add(valeur);
    }

  }

}
```

Avec la version 1.5, l'encapsulation de la valeur dans un objet n'est plus obligatoire car elle sera réalisée automatiquement par le compilateur.

Exemple (java 1.5) :

```
import java.util.*;

public class TestAutoboxing {

  public static void main(String[] args) {
    List liste = new ArrayList();
    for(int i = 0; i < 10; i++) {
      liste.add(i);
    }
  }

}
```

9.3. Les importations statiques

Jusqu'à la version 1.4 de Java, pour utiliser un membre statique d'une classe, il faut obligatoirement préfixer ce membre par le nom de la classe qui le contient.

Par exemple, pour utiliser la constante Pi définie dans la classe java.lang.Math, il est nécessaire d'utiliser Math.PI

Exemple :

```
public class TestStaticImportOld {

  public static void main(String[] args) {
    System.out.println(Math.PI);
    System.out.println(Math.sin(0));
  }

}
```

Java 1.5 propose une solution pour réduire le code à écrire concernant les membres statiques en proposant une nouvelle fonctionnalité concernant l'importation de package : l'import statique (static import).

Ce nouveau concept permet d'appliquer les mêmes règles aux membres statiques qu'aux classes et interfaces pour l'importation classique.

Cette nouvelle fonctionnalité est développée dans la JSR 201. Elle s'utilise comme une importation classique en ajoutant le mot clé static.

Exemple (java 1.5) :

```
import static java.lang.Math.*;

public class TestStaticImport {

  public static void main(String[] args) {
    System.out.println(PI);
    System.out.println(sin(0));
  }

}
```

L'utilisation de l'importation statique s'applique à tous les membres statiques : constantes et méthodes statiques de l'élément importé.

9.4. Les annotations ou méta données (Meta Data)

Cette nouvelle fonctionnalité est spécifiée dans la JSR 175.

Elle propose de standardiser l'ajout d'annotations dans le code. Ces annotations pourront ensuite être traitées par des outils pour générer d'autres éléments tels que des fichiers de configuration ou du code source.

Ces annotations concernent les classes, les méthodes et les champs. Leurs syntaxes utilisent le caractère « @ ».

La mise en oeuvre détaillée des annotations est proposée dans le chapitre qui leur est consacrés.

9.5. Les arguments variables (varargs)

Cette nouvelle fonctionnalité va permettre de passer un nombre non défini d'arguments d'un même type à une méthode. Ceci va éviter de devoir encapsuler ces données dans une collection.

Cette nouvelle fonctionnalité est spécifiée dans la JSR 201. Elle implique une nouvelle notation pour préciser la répétition d'un type d'argument. Cette nouvelle notation utilise trois petits points : ...

Exemple (java 1.5) :

```
public class TestVarargs {

  public static void main(String[] args) {
    System.out.println("valeur 1 = " + additionner(1,2,3));
    System.out.println("valeur 2 = " + additionner(2,5,6,8,10));
  }

  public static int additionner(int ... valeurs) {
        int total = 0;

        for (int val : valeurs) {
                total += val;
        }

        return total;
  }
```

```
}
```

Résultat :

```
C:\tiger>java TestVarargs
valeur 1 = 6
valeur 2 = 31
```

L'utilisation de la notation ... permet le passage d'un nombre indéfini de paramètres du type précisé. Tous ces paramètres sont traités comme un tableau : il est d'ailleurs possible de fournir les valeurs sous la forme d'un tableau.

Exemple (java 1.5) :

```
public class TestVarargs2 {

  public static void main(String[] args) {
    int[] valeurs = {1,2,3,4};
    System.out.println("valeur 1 = " + additionner(valeurs));
  }

  public static int additionner(int ... valeurs) {
        int total = 0;

        for (int val : valeurs) {
                total += val;
        }

        return total;
  }

}
```

Résultat :

```
C:\tiger>java TestVarargs2
valeur 1 = 10
```

Il n'est cependant pas possible de mixer des éléments unitaires et un tableau dans la liste des éléments fournis en paramètres.

Exemple (java 1.5) :

```
public class TestVarargs3 {

  public static void main(String[] args) {
    int[] valeurs = {1,2,3,4};
    System.out.println("valeur 1 = " + additionner(5,6,7,valeurs));
  }

  public static int additionner(int ... valeurs) {
        int total = 0;

        for (int val : valeurs) {
                total += val;
        }

        return total;
  }

}
```

Résultat :

```
C:\tiger>javac -source 1.5 -target 1.5 TestVarargs3.java
TestVarargs3.java:7: additionner(int[]) in TestVarargs3 cannot be applied to (in
t,int,int,int[])
```

```
    System.out.println("valeur 1 = " + additionner(5,6,7,valeurs));
                                       ^
1 error
```

9.6. Les generics

Les generics permettent d'accroître la lisibilité du code et surtout de renforcer la sécurité du code grâce à un renforcement du typage. Ils permettent de préciser explicitement le type d'un objet et rendent le cast vers ce type implicite. Cette nouvelle fonctionnalité est spécifiée dans la JSR 14.

Ils permettent par exemple de spécifier quel type d'objets une collection peut contenir et ainsi éviter l'utilisation d'un cast pour obtenir un élément de la collection.

L'inconvénient majeur du cast est que celui-ci ne peut être vérifié qu'à l'exécution et qu'il peut échouer. Avec l'utilisation des generics, le compilateur pourra réaliser cette vérification lors de la phase de compilation : la sécurité du code est ainsi renforcée.

Exemple (java 1.5) :

```
import java.util.*;

public class TestGenericsOld {

  public static void main(String[] args) {

    List liste = new ArrayList();
    String valeur = null;
    for(int i = 0; i < 10; i++) {
      valeur = ""+i;
      liste.add(valeur);
    }

    for (Iterator iter = liste.iterator(); iter.hasNext(); ) {
      valeur = (String) iter.next();
      System.out.println(valeur.toUpperCase());
    }
  }

}
```

L'utilisation des generics va permettre au compilateur de faire la vérification au moment de la compilation est de s'assurer ainsi qu'elle s'exécutera correctement. Ce mécanisme permet de s'assurer que les objets contenus dans la collection seront homogènes.

La syntaxe pour mettre en oeuvre les generics utilise les symboles < et > pour préciser le ou les types des objets à utiliser. Seuls des objets peuvent être utilisés avec les generics : si un type primitif est utilisé dans les generics, une erreur de type « unexpected type » est générée lors de la compilation.

Exemple (java 1.5) :

```
import java.util.*;

public class TestGenerics {

  public static void main(String[] args) {

    List<String> liste = new ArrayList();
    String valeur = null;
    for(int i = 0; i < 10; i++) {
      valeur = ""+i;
      liste.add(valeur);
    }
```

```
    for (Iterator<String> iter = liste.iterator(); iter.hasNext(); ) {
      System.out.println(iter.next().toUpperCase());
    }
  }

}
```

Si un objet de type différent de celui déclaré dans le generics est utilisé dans le code, le compilateur émet une erreur lors de la compilation.

Exemple (java 1.5) :

```
import java.util.*;

public class TestGenerics2 {

  public static void main(String[] args) {

    List<String> liste = new ArrayList();
    String valeur = null;
    for(int i = 0; i < 10; i++) {
      valeur = new Date();
      liste.add(valeur);
    }

    for (Iterator<String> iter = liste.iterator(); iter.hasNext(); ) {
      System.out.println(iter.next().toUpperCase());
    }
  }

}
```

Résultat :

```
C:\tiger>javac -source 1.5 -target 1.5 TestGenerics2.java
TestGenerics2.java:10: incompatible types
found   : java.util.Date
required: java.lang.String
      valeur = new Date();
               ^
Note: TestGenerics2.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
1 error
```

L'utilisation des generics permet de rendre le code plus lisible et plus sûr notamment car il n'est plus nécessaire d'utiliser un cast et de définir une variable intermédiaire.

Les generics peuvent être utilisés avec trois éléments :

- Les classes
- Les interfaces
- Les méthodes

Pour définir une classe utilisant les generics, il suffit de déclarer leur utilisation dans la signature de la classe à l'aide des caractères < et >. Ce type de déclaration est appelé type paramétré (parameterized type) Dans ce cas, les paramètres fournis dans la déclaration du generics sont des variables de types. Si la déclaration possède plusieurs variables de type alors il faut les séparer par un caractère virgule.

Exemple (java 1.5) :

```
public class MaClasseGeneric<T1, T2> {
  private T1 param1;
  private T2 param2;

  public MaClasseGeneric(T1 param1, T2 param2) {
```

```
    this.param1 = param1;
    this.param2 = param2;
  }

  public T1 getParam1() {
    return this.param1;
  }

  public T2 getParam2() {
    return this.param2;
  }
}
```

Lors de l'utilisation de la classe, il faut utiliser les types paramétrés pour indiquer le type des objets à utiliser.

Exemple (java 1.5) :

```
import java.util.*;

public class TestClasseGeneric {

  public static void main(String[] args) {
    MaClasseGeneric<Integer, String> maClasse =
          new MaClasseGeneric<Integer, String>(1, "valeur 1");
    Integer param1 = maClasse.getParam1();
    String  param2 = maClasse.getParam2();

  }

}
```

Le principe est identique avec les interfaces.

La syntaxe utilisant les caractères < et > se situe toujours après l'entité qu'elle concerne.

Exemple (java 1.5) :

```
    MaClasseGeneric<Integer, String> maClasse =
                new MaClasseGeneric<Integer, String>(1, "valeur 1");
    MaClasseGeneric<Integer, String>[] maClasses;
```

Même le cast peut être utilisé avec le generics en utilisant le nom du type paramétré dans le cast.

Il est possible de préciser une relation entre une variable de type et une classe ou interface : ainsi il sera possible d'utiliser une instance du type paramétré avec n'importe quel objet qui hérite ou implémente la classe ou l'interface précisée avec le mot clé extend dans la variable de type.

Exemple (java 1.5) :

```
import java.util.*;

public class MaClasseGeneric2<T1 extends Collection> {
  private T1 param1;

  public MaClasseGeneric2(T1 param1) {
    this.param1 = param1;
  }

  public T1 getParam1() {
    return this.param1;
  }

}
```

L'utilisation du type paramétré MaClasseGeneric2 peut être réalisée avec n'importe quelle classe qui hérite de l'interface java.util.Collection.

Exemple (java 1.5) :

```
import java.util.*;

public class TestClasseGeneric2 {

  public static void main(String[] args) {
    MaClasseGeneric2<ArrayList> maClasseA =
          new MaClasseGeneric2<ArrayList>(new ArrayList());
    MaClasseGeneric2<TreeSet> maClasseB =
          new MaClasseGeneric2<TreeSet>(new TreeSet());
  }

}
```

Ce mécanisme permet une utilisation un peu moins strict du typage dans les generics.

L'utilisation d'une classe qui n'hérite pas de la classe où n'implémente pas l'interface définie dans la variable de type, provoque une erreur à la compilation.

Exemple (java 1.5) :

```
C:\tiger>javac -source 1.5 -target 1.5 TestClasseGeneric2.java
TestClasseGeneric2.java:8: type parameter java.lang.String is not within its bou
nd
    MaClasseGeneric2<String> maClasseC = new MaClasseGeneric2<String>("test");
                     ^
TestClasseGeneric2.java:8: type parameter java.lang.String is not within its bou
nd
    MaClasseGeneric2<String> maClasseC = new MaClasseGeneric2<String>("test");
                                                              ^
2 errors
```

9.7. Les boucles pour le parcours des collections

L'itération sur les éléments d'une collection est fastidieuse avec la déclaration d'un objet de type Iterator.

Exemple :

```
import java.util.*;

public class TestForOld {

  public static void main(String[] args) {
    List liste = new ArrayList();
    for(int i = 0; i < 10; i++) {
      liste.add(i);
    }

    for (Iterator iter = liste.iterator(); iter.hasNext(); ) {
      System.out.println(iter.next());
    }
  }

}
```

La nouvelle forme de l'instruction for, spécifiée dans la JSR 201, permet de simplifier l'écriture du code pour réaliser une telle itération et laisse le soin au compilateur de générer le code nécessaire.

Exemple (java 1.5) :

```
import java.util.*;

public class TestFor {

  public static void main(String[] args) {
    List liste = new ArrayList();
    for(int i = 0; i < 10; i++) {
      liste.add(i);
    }

    for (Object element : liste) {
      System.out.println(element);
    }
  }

}
```

L'utilisation de la nouvelle syntaxe de l'instruction for peut être renforcée en combinaison avec les generics, ce qui évite l'utilisation d'un cast.

Exemple (java 1.5) :

```
import java.util.*;
import java.text.*;

public class TestForGenerics {

  public static void main(String[] args) {
    List<Date> liste = new ArrayList();
    for(int i = 0; i < 10; i++) {
      liste.add(new Date());
    }

    DateFormat df = DateFormat.getDateInstance();

    for (Date element : liste) {
      System.out.println(df.format(element));
    }
  }

}
```

La nouvelle syntaxe de l'instruction peut aussi être utilisée pour parcourir tous les éléments d'un tableau.

Exemple (java 1.5) :

```
import java.util.*;

public class TestForArray {

  public static void main(String[] args) {
    int[] tableau = {0,1,2,3,4,5,6,7,8,9};

    for (int element : tableau) {
      System.out.println(element);
    }
  }

}
```

L'exemple précédent fait aussi usage d'une autre nouvelle fonctionnalité du JDK 1.5 : l'unboxing.

Cela permet d'éviter la déclaration et la gestion dans le code d'une variable contenant l'index courant lors du parcours du tableau.

9.8. Les énumérations (type enum)

Souvent lors de l'écriture de code, il est utile de pouvoir définir un ensemble fini de valeurs pour une donnée pour par exemple définir les valeurs possibles qui vont caractériser l'état de cette donnée.

Pour cela, le type énumération permet de définir un ensemble de constantes : une énumération est un ensemble fini d'éléments constants. Cette fonctionnalité existe déjà dans les langages C et Delphi entre autre.

Jusqu'à la version 1.4 incluse, la façon la plus pratique pour palier au manque du type enum était de créer des constantes dans une classe.

Exemple :

```
public class FeuTricolore {
  public static final int VERT = 0;
  public static final int ORANGE = 1;
  public static final int ROUGE = 1;
}
```

Cette approche fonctionne : les constantes peuvent être sérialisées et utilisées dans une instruction switch mais leur mise en oeuvre n'est pas type safe. Rien n'empêche d'affecter une autre valeur à la donnée de type int qui va stocker une des valeurs constantes.

A défaut de mieux, cette solution permet de répondre au besoin mais elle possède cependant quelques inconvénients :

- le principal inconvénient de cette technique est qu'il n'y a pas de contrôle sur la valeur affectée à une donnée surtout si les constantes ne sont pas utilisées : il est possible d'utiliser n'importe quelle valeur permise par le type de la variable en plus des constantes définies. Le compilateur ne peut faire aucun contrôle sur les valeurs utilisées
- il n'est pas possible de faire une itération sur chacune des valeurs
- il n'est pas possible d'associer des traitements à une constante
- les modifications faites dans ces constantes notamment les changements de valeurs ne sont pas automatiquement reportées dans les autres classes qui doivent être explicitement recompilées

Java 5 apporte un nouveau type nommé enum qui permet de définir un ensemble de champs constants. Cette nouvelle fonctionnalité est spécifiée dans la JSR 201.

Un exemple classique est l'énumération des jours de la semaine.

Exemple :

```
public enum Jour {
  LUNDI, MARDI, MERCREDI, JEUDI,
  VENDREDI, SAMEDI, DIMANCHE
}
```

Les énumérations permettent de définir un ensemble fini de constantes, chacunes d'entres elles est séparées des autres par une virgule. Comme ces champs sont constants, leur nom est en majuscule par convention.

9.8.1. La définition d'une énumération

La définition d'une énumération ressemble à celle d'une classe avec quelques différences :

- utilisation du mot clé enum introduit spécifiquement dans ce but à la place du mot clé class
- un ensemble de valeurs constantes définies au début du corps de la définition, chaque valeur étant séparée par une virgule

* par convention le nom des constantes est en majuscule

Exemple :

```
public enum FeuTricolore {
  VERT, ORANGE, ROUGE
}
```

Une énumération peut prendre plusieurs formes et être enrichies de fonctionnalités puisqu'une énumération est une classe Java.

Dans sa forme la plus simple, la déclaration d'une énumération se résume à définir l'ensemble des constantes.

Exemple :

```
public enum FeuTricolore {
  VERT, ORANGE, ROUGE;
};
```

Les énumérations peuvent être déclarées à plusieurs niveaux. Le mot clé enum est au même niveau que le mot clé class ou interface : une énumération peut donc être déclarée au même endroit qu'une classe ou une interface, que cela soit dans un fichier dédié ou dans le fichier d'une autre classe.

Exemple :

```
public class TestEnum2 {
  public enum MonStyle {
    STYLE_1, STYLE_2, STYLE_3, STYLE_4, STYLE_5
  };

  public static void main(String[] args) {
    afficher(MonStyle.STYLE_2);
  }

  public static void afficher(MonStyle style) {
    System.out.println(style);
  }
}
```

Lors de la compilation de cet exemple, une classe interne est créée pour encapsuler l'énumération.

Résultat :

```
C:\java\workspace\TestEnum\bin>dir
 Le volume dans le lecteur C s'appelle Disque_C
 Le numéro de série du volume est 043F-2ED6

 Répertoire de C:\java\workspace\TestEnum\bin

15/07/2010  16:33    <REP>          .
15/07/2010  16:33    <REP>          ..
15/07/2010  16:39             1 160 TestEnum2$MonStyle.class
15/07/2010  16:39               743 TestEnum2.class
               2 fichier(s)            1 903 octets
               2 Rép(s)  23 175 589 888 octets libres
```

Les modificateurs d'accès s'appliquent sur une énumération.

L'outil Javadoc recense les énumérations dans le fichier package-summary.html.

9.8.2. L'utilisation d'une énumération

Le nom utilisé dans la déclaration de l'énumération peut être utilisé comme n'importe quelle classe dans la déclaration d'un type.

Une fois définie, il est possible d'utiliser l'énumération simplement en définissant une variable du type de l'énumération.

Exemple :

```
public class TestEnum {
  Jour jour;

  public TestEnum(Jour jour) {
    this.jour = jour;
  }

  public void afficherJour() {
    switch (jour) {
    case LUNDI:
      System.out.println("Lundi");
      break;
    case MARDI:
      System.out.println("Mardi");
      break;
    case MERCREDI:
      System.out.println("Mercredi");
      break;
    case JEUDI:
      System.out.println("Jeudi");
      break;
    case VENDREDI:
      System.out.println("Vendredi");
      break;
    case SAMEDI:
      System.out.println("Samedi");
      break;
    case DIMANCHE:
      System.out.println("Dimanche");
      break;
    }
  }
  public static void main(String[] args) {
    TestEnum testEnum = new
    TestEnum(Jour.SAMEDI);
    testEnum.afficherJour();
  }
}
```

Lors de l'utilisation d'une constante, son nom doit être préfixé par le nom de l'énumération sauf dans le cas de l'utilisation dans une instruction switch.

Les énumérations étant transformées en une classe par le compilateur, il y a une vérification de type faite de l'utilisation de l'énumération à la compilation.

L'instruction switch a été modifiée pour permettre de l'utiliser avec une énumération puisque bien qu'étant physiquement une classe, celle-ci possède une liste finie de valeurs associées.

L'utilisation d'une énumération dans l'instruction switch impose de n'utiliser que le nom de la valeur sans la préfixer par le nom de l'énumération sinon une erreur est émise par le compilateur.

Exemple :

```
    switch(feu) {
    case (FeuTricolore.VERT) :
      System.out.println("passer");
      break;
    default :
      System.out.println("arreter");
```

```
        break;
    }
```

Résultat :

```
Feu.java:24: an enum switch case label must be the unqualified name of an enumeration constant
```

Chaque élément d'une énumération est associé à une valeur par défaut, qui débute à zéro et qui est incrémentée de un en un. La méthode ordinal() permet d'obtenir cette valeur.

Exemple :

```
    FeuTricolore feu = FeuTricolore.VERT;
    System.out.println(feu.ordinal());
```

Il y a plusieurs avantages à utiliser les enums à la place des constantes notamment le typage fort et le préfixe de la constante par l'énumération.

9.8.3. L'enrichissement de l'énumération

Une énumération peut mettre en oeuvre la plupart des fonctionnalités et des comportements d'une classe :

- implémenter une ou plusieurs interfaces
- avoir plusieurs constructeurs
- avoir des champs et des méthodes
- avoir un bloc d'initialisation statique
- avoir des classes internes (inner classes)

Un type enum hérite implicitement de la classe java.lang.Enum : il ne peut donc pas hériter d'une autre classe.

Exemple :

```
public enum MonEnum extends Object {
  UN, DEUX, TROIS;
}
```

Résultat :

```
MyType.java:3: '{' expected
public enum MonEnum extends Object {
MonEnum.java:6: expected
2 errors
```

Chacun des éléments de l'énumération est instancié via le constructeur sous la forme d'un champ public static.

Si les éléments de l'énumération sont définis sans argument alors un constructeur sans argument doit être proposé dans la définition de l'énumération (celui ci peut être le constructeur par défaut si aucun autre constructeur n'est défini).

Le fait qu'une énumération soit une classe permet de définir une espace de nommage pour ses éléments qui évite les collisions, par exemple Puissance.ELEVEE et Duree.ELEVEE.

A partir du code source de l'énumération, le compilateur va générer une classe enrichie avec certaines fonctionnalités.

Exemple :

```
C:\Users\Jean Michel\workspace\TestEnum\bin>javap FeuTricolore
Compiled from "FeuTricolore.java"
public final class FeuTricolore extends java.lang.Enum{
```

```
    public static final FeuTricolore VERT;
    public static final FeuTricolore ORANGE;
    public static final FeuTricolore ROUGE;
    static {};
    public static FeuTricolore[] values();
    public static FeuTricolore valueOf(java.lang.String);
}
```

La classe compilée a été enrichie automatiquement par le compilateur qui a identifié l'entité comme une énumération grace au mot clé enum :

- la classe compilée hérite de la classe java.lang.Enum.
- un champ public static final du type de l'énumération est ajouté pour chaque élément
- un bloc d'initialisation static permet de créer les différentes instances statiques des éléments
- les méthodes valueOf() et values() sont ajoutées

Le compilateur ajoute automatiquement certaines méthodes à une classe de type enum lors de la compilation, notamment les méthodes statiques :

- values() qui renvoie un tableau des valeurs de l'énumération
- valueOf() qui renvoie l'élément de l'énumération dont le nom est fourni en paramètre

Le nom fourni en paramètre de la méthode valueOf() doit correspondre exactement à l'identifiant utilisé dans la déclaration de l'énumération. Il n'est pas possible de redéfinir la méthode valueOf().

Une énumération propose une implémentation par défaut de la méthode toString() : par défaut, elle renvoie le nom de la constante. Il est possible de la rédéfinir aux besoins.

Il est possible de préciser une valeur pour chaque élément de l'énumération lors de sa définition : celle ci sera alors stockée et pour être utilisée dans les traitements.

Exemple :

```
public enum Coefficient {
  UN(1), DEUX(2), QUATRE(4);

  private final int valeur;

  private Coefficient(int valeur) {
    this.valeur = valeur;
  }

  public int getValeur() {
    return this.valeur;
  }
}
```

Dans ce cas, l'énumération doit implicitement définir :

- un attribut qui contient la valeur associée à l'élément
- un constructeur qui attend en paramètre la valeur
- une méthode de type getter pour obtenir la valeur

Attention : toutes les données manipulées dans un élément d'une énumération doivent être immuables. Par exemple, il ne faut pas encapsuler dans un élément d'une énumération une donnée dont la valeur peut fluctuer dans le temps puisque l'élément est instancié une seule et unique fois.

Il faut obligatoirement définir les constantes en premier, avant toute définition de champs ou de méthodes. Si l'enum contient des champs et/ou des méthodes, il est impératif de terminer la définition des contantes par un point virgule.

Il est aussi possible de fournir plusieurs valeurs à un élément de l'énumération : comme une énumération est une classe, il est possible d'associer plusieurs valeurs à un élément de l'énumération. Ces valeurs seront stockées sous la forme de

propriétés et l'énumération doit fournir un constructeur qui doit accepter en paramètre les valeurs de chaque propriété.

Exemple :

```
import java.math.BigDecimal;

public enum Remise {

  COURANTE(new BigDecimal("0.05"), "Remise de 5%"),
  FIDELITE(new BigDecimal("0.07"), "Remise de 7%"),
  EXCEPTIONNELLE(new BigDecimal("0.10"), "Remise de 10%");

  private final BigDecimal taux;
  private final String libelle;

  private Remise(BigDecimal taux, String libelle) {
    this.taux = taux;
    this.libelle = libelle;
  }

  public BigDecimal getTaux() {
    return this.taux;
  }

  public String getLibelle() {
    return this.libelle;
  }

  public BigDecimal calculer(BigDecimal valeur) {
    return valeur.multiply(taux).setScale(2, BigDecimal.ROUND_FLOOR);
  }

  public static void main(String[] args) {
    BigDecimal montant = new BigDecimal("153.99");

    for (Remise remise : Remise.values()) {
      System.out.println(remise.getLibelle() + " \t"
        + remise.calculer(montant));
    }
  }
}
```

Résultat :

```
Remise de 5%   7.69
Remise de 7%   10.77
Remise de 10%  15.39
```

Dans l'exemple précédent, chaque constante est définie avec les deux paramètres qui la compose : le taux et le libellé. Ces valeurs sont passées au constructeur par le bloc d'initialisation static qui est créé par le compilateur.

Le constructeur d'une classe de type enum ne peut pas être public car il ne doit être invoqué que par la classe elle même pour créer les constantes définies au début de la déclaration de l'énumération.

Un élément d'une énumération ne peut avoir que la valeur avec laquelle elle est définie dans sa déclaration. Ceci justifie que le constructeur ne soit pas public et qu'une énumération ne puisse pas avoir de classes filles.

Tous les éléments de l'énumération sont encapsulés dans une instance finale du type de l'énumération : ce sont donc des singletons. De plus, les valeurs peuvent être testées avec l'opérateur == puisqu'elles sont déclarées avec le modificateur final.

Plusieurs fonctionnalités permettent de s'assurer qu'il n'y aura pas d'autres instances que celles définies par le compilateur à partir du code source :

- il n'y a pas de constructeur public qui permette de l'invoquer pour créer une nouvelle instance
- il n'est pas possible d'hériter d'une autre classe que la classe Enum
- il n'est pas possible de créer une classe fille de l'énumération puisqu'elle est déclarée finale

* l'invocation de la méthode clone() de l'énumération lève une exception de type CloneNotSupportedException

Une énumération peut implémenter une ou plusieurs interfaces. Comme une énumération est une classe, elle peut aussi contenir une méthode main().

9.8.4. La personnalisation de chaque élément

Le type Enum de Java est plus qu'une simple liste de constantes car une énumeration est définie dans une classe. Une classe de type Enum peut donc contenir des champs et des méthodes dédiées.

Pour encore plus de souplesse, il est possible de définir chaque élément sous la forme d'une classe interne dans laquelle il est possible de fournir une implémentation particulière pour chaque élément.

Il est ainsi possible de définir explicitement pour chaque valeur le corps de la classe qui va l'encapsuler. Une telle définition est similaire à la déclaration d'une classe anonyme. Cette classe est implicitement une extension de la classe englobante. Il est ainsi possible de redéfinir une méthode définie dans l'énumération.

Exemple :

```
import java.math.BigDecimal;

public enum Remise {

  COURANTE(new BigDecimal("0.05"), "Remise de 5%") {
    @Override
    public String toString() {
      return "Remise 5%";
    }
  },
  FIDELITE(new BigDecimal("0.07"), "Remise de 7%") {
    @Override
    public String toString() {
      return "Remise fidelite 7%";
    }
  },
  EXCEPTIONNEL(new BigDecimal("0.10"), "Remise de 10%") {
    @Override
    public String toString() {
      return "Remise exceptionnelle 10%";
    }

    @Override
    public String getLibelle() {
      return "Remise à titre exceptionnelle de 10%";
    }
  };

  private final BigDecimal taux;
  private final String libelle;

  private Remise(BigDecimal taux, String libelle) {
    this.taux = taux;
    this.libelle = libelle;
  }

  public BigDecimal getTaux() {
    return this.taux;
  }

  public String getLibelle() {
    return this.libelle;
  }

  public BigDecimal calculer(BigDecimal valeur) {
    return valeur.multiply(taux).setScale(2, BigDecimal.ROUND_FLOOR);
  }

  public static void main(String[] args) {
```

```
    BigDecimal montant = new BigDecimal("153.99");

    for (Remise remise : Remise.values()) {
      System.out.println(remise + " \t" + remise.calculer(montant));
    }
  }
}
```

Résultat :

```
Remise 5%      7.69
Remise fidelite 7%     10.77
Remise exceptionnelle 10%      15.39
```

Il est aussi possible de définir une méthode abstract dans l'énumération pour forcer la définition de la méthode dans chaque élément.

Exemple :

```
import java.math.BigDecimal;

public enum Remise {

  COURANTE(new BigDecimal("0.05"), "Remise de 5%") {

    @Override
    public String getLibelle() {
      return this.libelle;
    }

  },
  FIDELITE(new BigDecimal("0.07"), "Remise de 7%") {

    @Override
    public String getLibelle() {
      return "Remise fidélité de 7%";
    }
  },
  EXCEPTIONNEL(new BigDecimal("0.10"), "Remise de 10%") {

    @Override
    public String getLibelle() {
      return "Remise à titre exceptionnelle de 10%";
    }
  };

  private final BigDecimal taux;
  protected final String libelle;

  private Remise(BigDecimal taux, String libelle) {
    this.taux = taux;
    this.libelle = libelle;
  }

  public BigDecimal getTaux() {
    return this.taux;
  }

  public abstract String getLibelle();

  public BigDecimal calculer(BigDecimal valeur) {
    return valeur.multiply(taux).setScale(2, BigDecimal.ROUND_FLOOR);
  }

  public static void main(String[] args) {
    BigDecimal montant = new BigDecimal("153.99");

    for (Remise remise : Remise.values()) {
      System.out.println(remise.getLibelle() + " \t" + remise.calculer(montant));
    }
```

```
  }
}
```

Résultat :

```
Remise de 5%   7.69
Remise fidélité de 7%  10.77
Remise à titre exceptionnelle de 10%  15.39
```

Il faut cependant utiliser cette possibilité avec parsymonie car le code est moins lisible.

9.8.5. Les limitations dans la mise en oeuvre des énumérations

La mise en oeuvre des énumérations possède plusieurs limitations.

L'ordre de définition du contenu de l'énumération est important : les éléments de l'énumération doivent être définis en premier.

Un élément d'une énumération ne doit pas être null.

Un type Enum hérite implictement de la classe java.lang.Enum : il ne peut pas hériter d'une autre classe mère.

Pour garantir qu'il n'y ait qu'une seule instance d'un élément d'une énumération, le constructeur n'est pas accessible et l'énumération ne peut pas avoir de classe fille.

Une énumération ne peut pas être définie localement dans une méthode.

La méthode values() renvoie un tableau des éléments de l'énumération dans l'ordre dans lequel ils sont déclarés mais il ne faut surtout pas utiliser l'ordre des éléments d'une énumération dans les traitements : il ne faut par exemple pas tester la valeur retournée par la méthode ordinal() dans les traitements. Des problèmes apparaitront à l'éxécution si l'ordre des éléments est modifié car le compilateur ne peut pas détecter ce type de soucis.

Il n'est pas possible de personnaliser la sérialisation d'une énumération en redéfinissant les méthodes writeObject() et writeReplace() qui seront ignorées lors de la sérialisation. De plus, la déclaration d'un serialVersionUID est ignorée car sa valeur est toujours 0L.

10. Les annotations

Java SE 5 a introduit les annotations qui sont des métas données incluses dans le code source. Les annotations ont été spécifiées dans la JSR 175 : leur but est d'intégrer au langage Java des métas données.

Des métas données étaient déjà historiquement mises en oeuvre avec Java notamment avec Javadoc ou exploitées par des outils tiers notamment XDoclet : l'outil open source XDoclet propose depuis déjà longtemps des fonctionnalités similaires aux annotations. Avant Java 5, seul l'outil Javadoc utilisait des métas données en standard pour générer une documentation automatique du code source.

Javadoc propose l'annotation @deprecated qui bien qu'utilisé dans les commentaires permet de marquer une méthode comme obsolète et faire afficher un avertissement par le compilateur.

Le défaut de Javadoc est d'être trop spécifique à l'activité de génération de documentation même si le tag deprecated est aussi utilisé par le compilateur.

Les annotations de Java 5 apportent une standardisation des métas données dans un but généraliste. Ces métas données associés aux entités Java peuvent être exploitées à la compilation ou à l'exécution.

Java a été modifié pour permettre la mise en oeuvre des annotations :

- une syntaxe dédiée a été ajoutée dans Java pour permettre la définition et l'utilisation d'annotations.
- le byte code est enrichi pour permettre le stockage des annotations.

Les annotations peuvent être utilisées avec quasiment tous les types d'entités et de membres de Java : packages, classes, interfaces, constructeurs, méthodes, champs, paramètres, variables ou annotations elles même.

Java 5 propose plusieurs annotations standards et permet la création de ces propres annotations.

Une annotation s'utilise en la faisant précéder de l'entité qu'elle concerne. Elle est désignée par un nom précédé du caractère @.

Il existe plusieurs catégories d'annotations :

- les marqueurs (markers) : (exemple : @Deprecated, @Override, ...)
- les annotations paramétrées (exemple : @MonAnnotation("test"))
- les annotations multi paramétrées : exemple : @MonAnnotation(arg1="test 3", arg2="test 2", arg3="test3")

Les arguments fournis en paramètres d'une annotation peuvent être de plusieurs types : les chaînes de caractères, les types primitifs, les énumérations, les annotations, le type Class.

Les annotations sont définies dans un type d'annotation. Une annotation est une instance d'un type d'annotation. Les annotations peuvent avoir des valeurs par défaut.

La disponibilité d'une annotation est définie grâce à une retention policy.

Les usages des annotations sont nombreux : génération de documentions, de code, de fichiers, ORM (Object Relational Mapping), ...

Les annotations ne sont guère utiles sans un mécanisme permettant leur traitement.

Une API est proposée pour assurer ces traitements : elle est regroupée dans les packages com.sun.mirror.apt, com.sun.mirror.declaration, com.sun.mirror.type et com.sun.mirror.util.

L'outil apt (annotation processing tool) permet un traitement des annotations personnalisées durant la phase de compilation (compile time). L'outil apt permet la génération de nouveaux fichiers mais ne permet pas de modifier le code existant.

Il est important de se souvenir que lors du traitement des annotations le code source est parcouru mais il n'est pas possible de modifier ce code.

L'API reflexion est enrichie pour permettre de traiter les annotations lors de la phase d'exécution (runtime).

Java 6 intègre deux JSR concernant les annotations :

- Pluggable Annotation Processing API (JSR 269)
- Common Annotations (JSR 250)

L'Api Pluggable Annotation Processing permet d'intégrer le traitement des annotations dans le processus de compilation du compilateur Java ce qui évite d'avoir à utiliser apt.

Les annotations vont évoluer dans la plate-forme Java notamment au travers de plusieurs JSR qui sont en cours de définition :

- JSR 305 Annotations for Software Defect Detection
- JSR 308 Annotations on Java Types : doit permettre de mettre en oeuvre les annotations sur tous les types notamment les generics et sur les variables locales à l'exécution.

10.1. La mise en oeuvre des annotations

Les annotations fournissent des informations sur des entités : elles n'ont pas d'effets directs sur les entités qu'elles concernent.

Les annotations utilisent leur propre syntaxe. Une annotation s'utilise avec le caractère @ suivi du nom de l'annotation : elle doit obligatoirement précéder l'entité qu'elle annote. Par convention, les annotations s'utilisent sur une ligne dédiée.

Les annotations peuvent s'utiliser sur les packages, les classes, les interfaces, les méthodes, les constructeurs et les paramètres de méthodes.

Exemple :

```
@Override
public void maMethode() {
}
```

Une annotation peut avoir un ou plusieurs attributs : ceux ci sont précisés entre parenthèses, séparés par une virgule. Un attribut est de la forme clé=valeur.

Exemple :

```
@SuppressWarnings(value = "unchecked")
void maMethode() { }
```

Lorsque l'annotation ne possède qu'un seul attribut, il est possible d'omettre son nom.

Exemple :

```
@SuppressWarnings("unchecked")
void maMethode() { }
```

Un attribut peut être de type tableau : dans ce cas, les différentes valeurs sont fournies entre accolade, chacune séparée par une virgule.

Exemple :

```
@SuppressWarnings(value={"unchecked", "deprecation"})
```

Le tableau peut contenir des annotations.

Exemple :

```
@TODOItems({
  @Todo(importance = Importance.MAJEUR,
      description = "Ajouter le traitement des erreurs",
      assigneA = "JMD",
      dateAssignation = "07-11-2007"),
  @Todo(importance = Importance.MINEURE,
      description = "Changer la couleur de fond",
      assigneA = "JMD",
      dateAssignation = "13-12-2007")
})
```

Il existe trois types d'annotations :

- Les marqueurs (markers) : ces annotations ne possèdent pas d'attribut
- Les annotations simples (single value annotations) : ces annotations ne possèdent qu'un seul attribut
- Les annotations complètes (full annotations) : ces annotations possèdent plusieurs attributs

10.2. L'utilisation des annotations

Les annotations prennent une place de plus en plus importantes dans la plate-forme Java et de nombreuses API open source.

Les utilisations des annotations concernent plusieurs fonctionnalités :

- Utilisation par le compilateur pour détecter des erreurs ou ignorer des avertissements
- Documentation
- Génération de code
- Génération de fichiers

10.2.1. La documentation

Les annotations peuvent être mise en oeuvre pour permettre la génération de documentations indépendantes de JavaDoc : liste de choses à faire, liste de services ou de composants, ...

Il peut par exemple être pratique de rassembler certaines informations mises sous la forme de commentaires dans des annotations pour permettre leur traitement.

Par exemple, il est possible de définir une annotation qui va contenir les métas données relatives aux informations sur une classe. Traditionnellement, une classe débute par un commentaire d'en-tête qui contient des informations sur l'auteur, la date de création, les modifications, ... L'idée est de fournir ces informations dans une annotation dédiée. L'avantage est de permettre facilement d'extraire et de manipuler ces informations qui ne seraient qu'informatives sous leur forme commentaires.

10.2.2. L'utilisation par le compilateur

Les trois annotations fournies en standard avec la plate-forme entre dans cette catégorie qui consiste à faire réaliser par le compilateur quelques contrôles basiques.

10.2.3. La génération de code

Les annotations sont particulièrement adaptées à la génération de code source afin de faciliter le travail des développeurs notamment sur des tâches répétitives.

Attention, le traitement des annotations ne peut pas modifier le code existant mais simplement créer de nouveaux fichiers sources.

10.2.4. La génération de fichiers

Les API standards ou les frameworks open source nécessitent fréquemment l'utilisation de fichiers de configuration ou de déploiement généralement au format XML.

Les annotations peuvent proposer une solution pour maintenir le contenu de ces fichiers par rapport aux entités incluses dans le code de l'application.

La version 5 de Java EE fait un important usage des annotations dans le but de simplifier les développements de certains composants notamment les EJB, les entités et les services web. Pour cela, l'utilisation de descripteurs est remplacée par l'utilisation d'annotations ce qui rend le code plus facile à développer et plus claire.

10.2.5. Les API qui utilisent les annotations

De nombreuses API standards utilisent les annotations depuis leur intégration dans Java notamment :

- JAXB 2.0 : JSR 222 (Java Architecture for XML Binding 2.0)
- Les services web de Java 6 (JAX-WS) : JSR 181 (Web Services Metadata for the Java Platform) et JSR 224 (Java APIs for XML Web Services 2.0 API)
- Les EJB 3.0 et JPA : JSR 220 (Enterprise JavaBeans 3.0)

De nombreuses API open source utilisent aussi les annotations notamment JUnit, TestNG, Hibernate, ...

10.3. Les annotations standards

Java 5 propose plusieurs annotations standards.

10.3.1. L'annotation @Deprecated

Cette annotation a un rôle similaire au tag de même nom de Javadoc.

C'est un marqueur qui précise que l'entité concernée est obsolète et qu'il ne faudrait plus l'utiliser. Elle peut être utilisée avec une classe, une interface ou un membre (méthode ou champ)

Exemple :

```
public class TestDeprecated {

  public static void main(
      String[] args) {
    MaSousClasse td = new MaSousClasse();
    td.maMethode();
  }
}

@Deprecated
class MaSousClasse {

  /**
   * Afficher un message de test
   * @deprecated methode non compatible
   */
  @Deprecated
  public void maMethode() {
    System.out.println("test");
  }
}
```

Les entités marquées avec l'annotation @Deprecated devrait être documentée avec le tag @deprecated de Javadoc en lui fournissant la raison de l'obsolescence et éventuellement l'entité de substitution.

Il est important de tenir compte de la casse : @Deprecated pour l'annotation et @deprecated pour Javadoc.

Lors de la compilation, le compilateur donne une information si une entité obsolète est utilisée.

Exemple :

```
C:\Documents and Settings\jmd\workspace\Tests>javac TestDeprecated.java
Note: TestDeprecated.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
```

L'option -Xlint :deprecation permet d'afficher le détail sur les utilisations obsolètes.

Exemple :

```
C:\Documents and Settings\jmd\workspace\Tests>javac -Xlint:deprecation TestDepre
cated.java
TestDeprecated.java:7: warning: [deprecation] MaSousClasse in unnamed package ha
s been deprecated
    MaSousClasse td = new MaSousClasse();
    ^
TestDeprecated.java:7: warning: [deprecation] MaSousClasse in unnamed package ha
s been deprecated
    MaSousClasse td = new MaSousClasse();
                          ^
TestDeprecated.java:8: warning: [deprecation] maMethode() in MaSousClasse has be
en deprecated
    td.maMethode();
      ^
3 warnings
```

Il est aussi possible d'utiliser l'option -deprecation de l'outil javac.

10.3.2. L'annotation @Override

Cette annotation est un marqueur utilisé par le compilateur pour vérifier la réécriture de méthode héritée.

@Override s'utilise pour annoter une méthode qui est une réécriture d'une méthode héritée. Le compilateur lève une erreur si aucune méthode héritée ne correspond.

Exemple :

```
@Override
public void maMethode() {
}
```

Son utilisation n'est pas obligatoire mais recommandée car elle permet de détecter certains problèmes.

Exemple :

```
public class MaClasseMere {

}

class MaClasse extends MaClasseMere {

  @Override
  public void maMethode() {

  }
}
```

Ceci est particulièrement utile pour éviter des erreurs de saisie dans le nom des méthodes à redéfinir.

Exemple :

```
public class TestOverride {
  private String nom;
  private long id;

  public int hasCode() {
    final int PRIME = 31;
    int result = 1;
    result = PRIME * result + (int) (id ^ (id >>> 32));
    result = PRIME * result + ((nom == null) ? 0 : nom.hashCode());
    return result;
  }
}
```

Cette classe se compile parfaitement mais elle comporte une erreur qui est signalée en utilisation l'annotation @Override

Exemple :

```
public class TestOverride {
  private String nom;
  private long id;

  @Override
  public int hasCode() {
    final int PRIME = 31;
    int result = 1;
    result = PRIME * result + (int) (id ^ (id >>> 32));
    result = PRIME * result + ((nom == null) ? 0 : nom.hashCode());
    return result;
  }
}
```

Résultat :

```
C:\Documents and Settings\jmd\workspace\Tests>javac TestOverride.java
TestOverride.java:6: method does not override or implement a method from a super
```

```
type
  @Override
  ^
1 error
```

10.3.3. L'annotation @SuppressWarning

L'annotation @SuppressWarning permet de demander au compilateur d'inhiber certains avertissements qui sont pris en compte par défaut.

La liste des avertissements utilisables dépend du compilateur. Un avertissement utilisé dans l'annotation non reconnu par le compilateur ne provoque par d'erreur mais éventuellement un avertissement.

Le compilateur fourni avec le JDK supporte les avertissements suivants :

Nom	Rôle
deprecation	Vérification de l'utilisation d'entités déclarées deprecated
unchecked	Vérification de l'utilisation des generics
fallthrough	Vérification de l'utilisation de l'instruction break dans les cases des instructions switch
path	Vérification des chemins fournis en paramètre du compilateur
serial	Vérification de la définition de la variable serialVersionUID dans les beans
finally	Vérification de la non présence d'une instruction return dans une clause finally

Il est possible de passer en paramètres plusieurs types d'avertissements sous la forme d'un tableau

Exemple :

```
@SuppressWarnings(value={"unchecked", "fallthrough"})
```

L'exemple ci-dessous génère un avertissement à la compilation

Exemple :

```
import java.util.ArrayList;
import java.util.List;

public class TestSuppresWarning {
  public static void main(
      String[] args) {
    List donnees = new ArrayList();
    donnees.add("valeur1");
  }
}
```

Résultat :

```
C:\Documents and Settings\jmd\workspace\Tests>javac TestSuppresWarning.java
Note: TestSuppresWarning.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

L'option -Xlint :unchecked permet d'obtenir des détails

Exemple :

```
C:\Documents and Settings\jmd\workspace\Tests>javac -Xlint:unchecked TestSuppres
Warning.java
TestSuppresWarning.java:8: warning: [unchecked] unchecked call to add(E) as a me
mber of the raw type java.util.List
    donnees.add("valeur1");
                ^
1 warning
```

Pour supprimer cet avertissement, il faut utiliser les générics dans la déclaration de la collection ou utiliser l'annotation SuppressWarning.

Exemple :

```
import java.util.ArrayList;
import java.util.List;

@SuppressWarnings("unchecked")
public class TestSuppresWarning {
  public static void main(
      String[] args) {
    List donnees = new ArrayList();
    donnees.add("valeur1");
  }
}
```

Il n'est pas recommandé d'utiliser cette annotation mais plutôt d'apporter une solution à l'avertissement.

10.4. Les annotations communes (Common Annotations)

Les annotations communes sont définies par la JSR 250 et sont intégrées dans Java 6. Leur but est de définir des annotations couramment utilisées et ainsi éviter leur redéfinition pour chaque outil qui en aurait besoin.

Les annotations définies concernent :

- la plate-forme standard dans le package javax.annotation (@Generated, @PostConstruct, @PreDestroy, @Resource, @Resources)
- la plate-forme entreprise dans le package javax.annotation.security (@DeclareRoles, @DenyAll, @PermitAll, @RolesAllowed, @RunAs).

10.4.1. L'annotation @Generated

De plus en plus d'outils ou de frameworks génèrent du code source pour faciliter la tâche des développeurs notamment pour des portions de code répétitives ayant peu de valeur ajoutée.

Le code ainsi généré peut être marqué avec l'annotation @Generated.

Exemple :

```
    @Generated(
        value = "entite.qui.a.genere.le.code",
        comments = "commentaires",
        date = "12 April  2008"
    )
    public void toolGeneratedCode(){
    }
```

L'attribut obligatoire value permet de préciser l'outil à l'origine de la génération

Les attributs facultatifs comments et date permettent respectivement de fournir un commentaire et la date de génération.

Cette annotation peut être utilisée sur toutes les déclarations d'entités.

10.4.2. Les annotations @Resource et @Resources

L'annotation @Resource définit une ressource requise par une classe. Typiquement, une ressource est par exemple un composant Java EE de type EJB ou JMS.

L'annotation @Resource possède plusieurs attributs :

Attribut	Description
authentificationType	Type d'authentification pour utiliser la ressource (Resource.AuthenticationType.CONTAINER ou Resource.AuthenticationType.APPLICATION)
description	Description de la ressource
mappedName	Nom de la ressource spécifique au serveur utilisé (non portable)
name	Nom JNDI de la ressource
shareable	Booléen qui précise si la ressource est partagée
type	Le type pleinement qualifié de la ressource

Cette annotation peut être utilisée sur une classe, un champ ou une méthode.

Lorsque l'annotation est utilisée sur une classe, elle correspond simplement à une déclaration des ressources qui seront requises à l'exécution.

Lorsque l'annotation est utilisée sur un champ ou une méthode, le serveur d'applications va injecter une référence sur la ressource correspondante. Pour cela, lors du chargement d'une application par le serveur d'applications, celui ci recherche les annotations @Resource afin d'assigner une instance de la ressource correspondante.

Exemple :

```
@Resource(name="MaQueue",
    type = "javax.jms.Queue",
    shareable=false,
    authenticationType=Resource.AuthenticationType.CONTAINER,
    description="Queue de test"
)
private javax.jms.Queue maQueue;
```

L'annotation @Resources est simplement une collection d'annotation de type @Resource.

Exemple :

```
@Resources({
    @Resource(name = "maQueue" type = javax.jms.Queue),
    @Resource(name = "monTopic" type = javax.jms.Topic),
})
```

10.4.3. Les annotations @PostConstruct et @PreDestroy

Les annotations @PostConstruct et @PreDestroy permettent respectivement de désigner des méthodes qui seront exécutées après l'instanciation d'un objet et avant la destruction d'un objet.

Ces deux annotations ne peuvent être utilisées que sur des méthodes.

Ces annotations sont par exemple utiles dans Java EE car généralement un composant géré par le conteneur est instancié en utilisant le constructeur sans paramètre. Une méthode marquée avec l'annotation @PostConstruct peut alors être exécutée juste après l'appel au constructeur.

Une telle méthode doit respecter certaines règles :

- ne pas voir de paramètres sauf dans des cas précis (exemple avec les intercepteurs des EJB)
- ne pas avoir de valeur de retour (elle doit renvoyer void)
- ne doit pas lever d'exceptions vérifiées
- ne doit pas être statique

L'annotation @PostConstruct est utilisée en général sur une méthode qui initialise des ressources en fonction du contexte.

Une seule méthode d'une même classe peut utiliser chacune de ces deux annotations.

10.5. Les annotations personnalisées

Java propose la possibilité de définir ces propres annotations. Pour cela, le langage possède un type dédié : le type d'annotation (annotation type).

Un type d'annotation est similaire à une classe et une annotation est similaire à une instance de classe.

10.5.1. La définition d'une annotation

Sur la plate-forme Java, une annotation est une interface lors de sa déclaration et est une instance d'une classe qui implémente cette interface lors de son utilisation.

La définition d'une annotation utilise une syntaxe particulière utilisant le mot clé @interface. Une annotation se déclare donc de façon similaire à une interface.

Exemple : le fichier MonAnnotation.java

```
package com.jmdoudoux.test.annotations;

public @interface MonAnnotation {

}
```

Une fois compilée, cette annotation peut être utilisée dans le code. Pour utiliser une annotation, il faut importer l'annotation et l'appeler dans le code en la faisant précéder du caractère @.

Exemple :

```
package com.jmdoudoux.test.annotations;

@MonAnnotation
public class MaCLasse {

}
```

Si l'annotation est définie dans un autre package, il faut utiliser la syntaxe pleinement qualifié du nom de l'annotation ou ajouter une clause import pour le package.

Il est possible d'ajouter des membres à l'annotation simplement en définissant une méthode dont le nom correspond au nom de l'attribut en paramètre de l'annotation.

Exemple :

```
package com.jmdoudoux.test.annotations;

public @interface MonAnnotation {
  String arg1();
  String arg2();
}

package com.jmdoudoux.test.annotations;

@MonAnnotation(arg1="valeur1", arg2="valeur2")
public class MaCLasse {

}
```

Les types utilisables sont les chaînes de caractères, les types primitifs, les énumérations, les annotations, les chaînes de caractères, le type Class.

Il est possible de définir un membre comme étant un tableau à une seule dimension d'un des types utilisables.

Exemple :

```
package com.jmdoudoux.tests.annotations

public @interface MonAnnotation {
    String arg1();
    String[] arg2();
    String arg3();
}
```

Il est possible de définir une valeur par défaut, ce qui les rend optionnels. Cette valeur est précisée en la faisant précéder du mot clé default.

Exemple :

```
package com.jmdoudoux.test.annotations;

public @interface MonAnnotation {
  String arg1() default "";
  String[] arg2();
  String arg3();
}
```

La valeur par défaut d'un tableau utilise une syntaxe raccourcie pour être précisée.

Exemple :

```
package com.jmdoudoux.tests.annotations

public @interface MonAnnotation {
    String arg1();
    String[] arg2() default {"chaine1", "chaine2" };
    String arg3();
}
```

Il est possible de définir une énumération comme type pour un attribut

Exemple :

```
package com.jmdoudoux.test.annotations;

public @interface MonAnnotation {
  public enum Niveau {DEBUTANT, CONFIRME, EXPERT} ;
  String arg1() default "";
  String[] arg2();
  String arg3();
  Niveau niveau() default Niveau.DEBUTANT;

}
```

10.5.2. Les annotations pour les annotations

La version 5 de Java propose quatre annotations dédiées aux types d'annotations qui permettent de fournir des informations sur l'utilisation d'une annotation.

Ces annotations sont définies dans le package java.lang.annotation

10.5.2.1. L'annotation @Target

L'annotation @Target permet de préciser les entités sur lesquelles l'annotation sera utilisable. Cette annotation attend comme valeur un tableau de valeurs issues de l'énumération ElementType

Valeur de l'énumération	Rôle
ANNOTATION_TYPE	Types d'annotation
CONSTRUCTOR	Constructeurs
LOCAL_VARIABLE	Variables locales
FIELD	Champs
METHOD	Méthodes hors constructeurs
PACKAGE	Packages
PARAMETER	paramètres d'une méthode ou d'un constructeur
TYPE	Classes, interfaces, énumérations, types d'annotation

Si une annotation est utilisée sur une entité non précisée par l'annotation, alors une erreur est émise lors de la compilation

Exemple :

```
package com.jmdoudoux.test.annotations;

import java.lang.annotation.ElementType;
import java.lang.annotation.Target;

@Target({ElementType.METHOD, ElementType.CONSTRUCTOR })
public @interface MonAnnotation {
  String arg1() default "";
  String arg2();
}

package com.jmdoudoux.test.annotations;

@MonAnnotation(arg1="valeur1", arg2="valeur2")
public class MaCLasse {
```

```
}
```

Résultat de la compilation :

```
C:\Documents and Settings\jmd\workspace\Tests>javac com/jmdoudoux/test/annotatio
ns/MaClasse.java
com\jmdoudoux\test\annotations\MaClasse.java:3: annotation type not applicable t
o this kind of declaration
@MonAnnotation(arg1="valeur1", arg2="valeur2")
^
1 error
```

10.5.2.2. L'annotation @Retention

Cette annotation permet de préciser à quel niveau les informations concernant l'annotation seront conservées. Cette annotation attend comme valeur un élément de l'énumération RetentionPolicy

Enumération	Rôle
RetentionPolicy.SOURCE	informations conservées dans le code source uniquement (fichier .java) : le compilateur les ignore
RetentionPolicy.CLASS	informations conservées dans le code source et le bytecode (fichier .java et .class)
RetentionPolicy.RUNTIME	informations conservées dans le code source et le bytecode et elles sont disponibles à l'exécution par introspection

Cette annotation permet de déterminer de quelle façon l'annotation pourra être exploitée.

Exemple :

```
@Retention(RetentionPolicy.RUNTIME)
```

10.5.2.3. L'annotation @Documented

L'annotation @Documented permet de demander l'intégration de l'annotation dans la documentation générée par Javadoc pour les entités annotées par cette annotation.

Par défaut, les annotations ne sont pas intégrées dans la documentation des classes annotées.

Exemple :

```
package com.jmdoudoux.test.annotations;

import java.lang.annotation.Documented;

@Documented
public @interface MonAnnotation {
  String arg1() default "";
  String arg2();
}
```

com.jmdoudoux.test.annotations
Class MaCLasse

```
java.lang.Object
  └ com.jmdoudoux.test.annotations.MaCLasse
```

```
@MonAnnotation(arg1="valeur1",
               arg2="valeur2")
public class MaCLasse
extends java.lang.Object
```

Author:
JMD

10.5.2.4. L'annotation @Inherited

L'annotation @Inherited permet de demander l'héritage d'une annotation aux classes filles de la classe mère sur laquelle elle s'applique.

Si une classe mère est annotée avec une annotation annotée avec @Inherited alors toutes les classes filles sont automatiquement annotées avec l'annotation.

10.6. L'exploitation des annotations

Pour être profitable les annotations ajoutées dans le code source doivent être exploitées par un ou plusieurs outils.

La déclaration et l'utilisation d'annotations sont relativement simples par contre leur exploitation pour permettre la production de fichiers est moins triviale.

Cette exploitation peut se faire de plusieurs manières

- en définissant un doclet qui exploite le code source
- en utilisant apt au moment de la compilation
- en utilisant introspection lors de l'exécution
- en utilisant le compilateur java à partir de Java 6.0

10.6.1. L'exploitation des annotations dans un Doclet

Pour des traitements simples, il est possible de définir un Doclet et de l'utiliser avec l'outil Javadoc pour utiliser les annotations.

L'API Doclet est définit dans le package com.sun.javadoc. Ce package est dans le fichier tools.jar fourni avec le JDK

L'API Doclet définit des interfaces pour chaque entités peuvent être utilisées dans le code source.

La méthode annotation() de l'interface ProgramElementDoc permet d'obtenir un tableau de type AnnotationDesc.

L'interface AnnotationDesc représente une annotation. Elle définit deux méthodes

Méthode	Rôle

AnnotationTypeDoc annotationType()	Renvoyer le type d'annotation
AnnotationDesc.ElementValuePair[] elementValues()	Renvoyer les éléments de l'annotation

L'interface AnnotationTypeDoc représente un type d'annotation. Elle ne définit qu'une seule méthode

Méthode	Rôle
AnnotationTypeElementDoc[] elements()	Renvoyer les éléments d'un type d'annotation

L'interface AnnotationTypeElementDoc représente un élément d'un type d'annotation Elle ne définit qu'une seule méthode

Méthode	Rôle
AnnotationValue defaultValue()	Renvoyer la valeur par défaut de l'élément d'un type d'annotation

L'interface AnnotationValue représente la valeur d'un élément d'un type d'annotation. Elle définit deux méthodes

Méthode	Rôle
String toString()	Renvoyer la valeur sous forme d'une chaîne de caractères
Object value()	Renvoyer la valeur

Pour créer un Doclet, il faut définir une classe qui contienne une méthode ayant pour signature `public static boolean start (RootDoc rootDoc).`

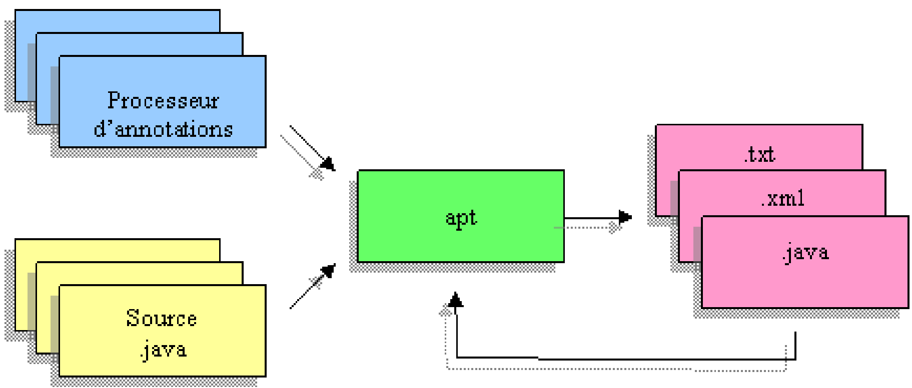
Pour utiliser le Doclet, il faut compiler la classe qui l'encapsule et utiliser l'outil javadoc avec l'option -doclet suivi du nom de la classe.

10.6.2. L'exploitation des annotations avec l'outil Apt

La version 5 du JDK fournit l'outil apt pour le traitement des annotations.

L'outil apt qui signifie annotation processing tool est l'outil le plus polyvalent en Java 5 pour exploiter les annotations.

Apt assure la compilation des classes et permet en simultané le traitement des annotations par des processeurs d'annotations créés par le développeur.



Les processeurs d'annotations peuvent générer de nouveaux fichiers source, pouvant eux même contenir des annotations. Apt traite alors récursivement les fichiers générés jusqu'à ce qu'il n'y ait plus d'annotations à traiter et de classes à compiler.

Cette section va créer un processeur pour l'annotation personnalisée Todo

Exemple : l'annotation personnalisée Todo

```
package com.jmdoudoux.test.annotations;

import java.lang.annotation.Documented;

@Documented
public @interface Todo {

  public enum Importance {
    MINEURE, IMPORTANT, MAJEUR, CRITIQUE
  };

  Importance importance() default Importance.MINEURE;

  String[] description();

  String assigneA();

  String dateAssignation();
}
```

Apt et l'API à utiliser de concert ne sont disponibles qu'avec le JDK : ils ne sont pas fournis avec le JRE.

Les packages de l'API sont dans le fichier lib/tools.jar du JDK : cette bibliothèque doit donc être ajoutée au classpath lors de la mise en oeuvre d'apt.

L'API est composée de deux grandes parties :

- Modélisation du langage
- Interaction avec l'outil de traitement des annotations

L'API est contenue dans plusieurs sous packages de com.sun.mirror notamment :

- com.sun.mirror.apt : contient les interfaces pour la mise en oeuvre d'apt
- com.sun.mirror.declaration : encapsule la déclaration des entités dans les sources qui peuvent être annotées (packages, classes, méthodes, ...) sous le forme d'interfaces qui héritent de l'interface Declaration
- com.sun.mirror.type : encapsule les types d'entités dans les sources sous la forme d'interfaces qui héritent de l'interface TypeMirror
- com.sun.mirror.util : propose des utilitaires

Un processeur d'annotations est une classe qui implémente l'interface com.sun.mirror.apt.AnnotationProcessor. Cette interface ne définit qu'une seule méthode process() qui va contenir les traitements à réaliser pour une annotation.

Il faut fournir un constructeur qui attend en paramètre un objet de type com.sun.mirror.apt.AnnotationProcessorEnvironment : ce constructeur sera appelé par une fabrique pour en créer une instance.

L'interface AnnotationProcessorEnvironment fournit des méthodes pour obtenir des informations sur l'environnement d'exécution des traitements des annotations et créer de nouveaux fichiers pendant les traitements.

L'interface Declaration permet d'obtenir des informations sur une entité :

Méthode	Rôle
<A extends Annotation> getAnnotation(Class<A> annotationType)	Renvoie une annotation d'un certain type associée à l'entité

Collection<AnnotationMirror> getAnnotationMirrors()	Renvoie les annotations associées à l'entité
String getDocComment()	Renvoie le texte des commentaires de documentations Javadoc associés à l'entité
Collection<Modifier> getModifiers()	Renvoie les modificateurs de l'entité
SourcePosition getPosition()	Renvoie la position de la déclaration dans le code source
String getSimpleName()	Renvoie le nom de la déclaration

De nombreuses interfaces héritent de l'interface Declaration : AnnotationTypeDeclaration, AnnotationTypeElementDeclaration, ClassDeclaration, ConstructorDeclaration, EnumConstantDeclaration, EnumDeclaration, ExecutableDeclaration, FieldDeclaration, InterfaceDeclaration, MemberDeclaration, MethodDeclaration, PackageDeclaration, ParameterDeclaration, TypeDeclaration, TypeParameterDeclaration

Chacune de ces interfaces propose des méthodes pour obtenir des informations sur la déclaration concernée.

L'interface TypeMirror permet d'obtenir des informations sur un type utilisé dans une déclaration.

De nombreuses interfaces héritent de l'interface TypeMirror : AnnotationType, ArrayType, ClassType, DeclaredType, EnumType, InterfaceType, PrimitiveType, ReferenceType, TypeVariable, VoidType, WildcardType.

Chacune de ces interfaces propose des méthodes pour obtenir des informations sur le type concerné.

La classe com.sun.mirror.util.DeclarationFilter permet de définir un filtre des entités annotées avec les annotations concernées par les traitements du processeur. Il suffit de créer une instance de cette classe en ayant redéfini sa méthode match(). Cette méthode renvoie un booléen qui précise si l'entité fournie en paramètre sous la forme d'un objet de type Declaration est annotée avec une des annotations concernées par le processeur.

Exemple :

```
package com.jmdoudoux.test.annotations.outils;

import java.util.Collection;

import com.jmdoudoux.test.annotations.Todo;
import com.sun.mirror.apt.AnnotationProcessor;
import com.sun.mirror.apt.AnnotationProcessorEnvironment;
import com.sun.mirror.declaration.Declaration;
import com.sun.mirror.declaration.TypeDeclaration;
import com.sun.mirror.util.DeclarationFilter;

public class TodoAnnotationProcessor implements AnnotationProcessor {
  private final AnnotationProcessorEnvironment env;

  public TodoAnnotationProcessor(AnnotationProcessorEnvironment env) {
    this.env = env;
  }

  public void process() {
    // Creation d'un filtre pour ne retenir que les déclarations annotées avec Todo
    DeclarationFilter annFilter = new DeclarationFilter() {
      public boolean matches(
          Declaration d) {
        return d.getAnnotation(Todo.class) != null;
      }
    };

    // Recherche des entités annotées avec Todo
    Collection<TypeDeclaration> types = annFilter.filter(env.getSpecifiedTypeDeclarations());
    for (TypeDeclaration typeDecl : types) {
      System.out.println("class name: " + typeDecl.getSimpleName());

      Todo todo = typeDecl.getAnnotation(Todo.class);

      System.out.println("description : ");
      for (String desc : todo.description()) {
```

```
        System.out.println(desc);
      }
      System.out.println("");
    }
  }
}
```

Il faut créer une fabrique de processeurs d'annotations : cette fabrique est en charge d'instancier des processeurs d'annotations pour un ou plusieurs types d'annotations. La fabrique doit implémenter l'interface com.sun.mirror.apt.AnnotationProcessorFactory.

L'interface AnnotationProcessorFactory déclare trois méthodes :

Méthode	Rôle
AnnotationProcessor getProcessorFor(Set<AnnotationTypeDeclaration> atds, AnnotationProcessorEnvironment env)	Renvoyer un processeur d'annotations pour l'ensemble de type d'annotations fournis en paramètres.
Collection<String> supportedAnnotationTypes()	Renvoyer une collection des types d'annotations dont un processeur peut être instancié par la fabrique
Collection<String> supportedOptions()	Renvoyer une collection des options supportées par la fabrique ou par les processeurs d'annotations créés par la fabrique

Exemple :

```
package com.jmdoudoux.test.annotations.outils;

import com.sun.mirror.apt.*;
import com.sun.mirror.declaration.*;

import java.util.Collection;
import java.util.Set;
import java.util.Collections;
import java.util.Arrays;

public class TodoAnnotationProcessorFactory implements AnnotationProcessorFactory {
  private static final Collection<String> supportedAnnotations =
    Collections.unmodifiableCollection(Arrays
      .asList("com.jmdoudoux.test.annotations.Todo"));

  private static final Collection<String> supportedOptions     = Collections.emptySet();

  public Collection<String> supportedOptions() {
    return supportedOptions;
  }

  public Collection<String> supportedAnnotationTypes() {
    return supportedAnnotations;
  }

  public AnnotationProcessor getProcessorFor(
      Set<AnnotationTypeDeclaration> atds,
      AnnotationProcessorEnvironment env) {
    return new TodoAnnotationProcessor(env);
  }
}
```

Dans l'exemple ci-dessus, aucune option n'est supportée et la fabrique ne prend en charge que l'annotation personnalisée Todo.

Pour mettre en oeuvre ces traitements des annotations, il faut que le code source utilise ces annotations.

Exemple : une classe annotée avec l'annotation Todo

```
package com.jmdoudoux.test;

import com.jmdoudoux.test.annotations.Todo;
import com.jmdoudoux.test.annotations.Todo.Importance;

@Todo(importance = Importance.CRITIQUE,
      description = "Corriger le bug dans le calcul",
      assigneA = "JMD",
      dateAssignation = "11-11-2007")
public class MaClasse {

}
```

Exemple : une autre classe annotée avec l'annotation Todo

```
package com.jmdoudoux.test;

import com.jmdoudoux.test.annotations.Todo;
import com.jmdoudoux.test.annotations.Todo.Importance;

@Todo(importance = Importance.MAJEUR,
      description = "Ajouter le traitement des erreurs",
      assigneA = "JMD",
      dateAssignation = "07-11-2007")
public class MaClasse1 {

}
```

Pour utiliser apt, il faut que le classpath contienne la bibliothèque tools.jar fournie avec le JDK et les classes de traitements des annotations (fabrique et processeur d'annotations).

L'option -factory permet de préciser la fabrique à utiliser.

Résultat de l'exécution d'apt

```
C:\Documents and Settings\jmd\workspace\Tests>apt -cp ".;./bin;C:/Program Files/
Java/jdk1.5.0_07/lib/tools.jar" -factory com.jmdoudoux.test.annotations.outils.T
odoAnnotationProcessorFactory com/jmdoudoux/test/*.java
class name: MaClasse
description :
Corriger le bug dans le calcul

class name: MaClasse1
description :
Ajouter le traitement des erreurs
```

A partir de l'objet de type AnnotationProcessorEnvironment, il est possible d'obtenir un objet de type com.sun.miror.apt.Filer qui encapsule un nouveau fichier créé par le processeur d'annotations.

L'interface Filer propose quatre méthodes pour créer différents types de fichiers :

Méthode	Rôle
OutputStream createBinaryFile(Filer.Location loc, String pkg, File relPath)	Créer un nouveau fichier binaire et renvoyer un objet de type Stream pour écrire son contenu
OutputStream createClassFile(String name)	Créer un nouveau fichier .class et renvoyer un objet de type Stream pour écrire son contenu
PrintWriter createSourceFile(String name)	Créer un nouveau fichier texte contenant du code source et renvoyer un objet de type Writer pour écrire son contenu

PrintWriter createTextFile(Filer.Location loc, String pkg, File relPath, String charsetName)	Créer un nouveau fichier texte et renvoyer un objet de type Writer pour écrire son contenu

L'énumération Filter.Location permet de préciser si le nouveau fichier est créé dans la branche source (SOURCE_TREE) ou dans la branche compilée (CLASS_TREE).

Exemple :

```
package com.jmdoudoux.test.annotations.outils;

import java.io.File;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Collection;

import com.jmdoudoux.test.annotations.Todo;
import com.sun.mirror.apt.AnnotationProcessor;
import com.sun.mirror.apt.AnnotationProcessorEnvironment;
import com.sun.mirror.apt.Filer;
import com.sun.mirror.declaration.Declaration;
import com.sun.mirror.declaration.TypeDeclaration;
import com.sun.mirror.util.DeclarationFilter;

public class TodoAnnotationProcessor implements AnnotationProcessor {
  private final AnnotationProcessorEnvironment env;

  public TodoAnnotationProcessor(AnnotationProcessorEnvironment env) {
    this.env = env;
  }

  public void process() {
    // Creation d'un filtre pour ne retenir que les déclarations annotées avec
    // Todo
    DeclarationFilter annFilter = new DeclarationFilter() {
      public boolean matches(
          Declaration d) {
        return d.getAnnotation(Todo.class) != null;
      }
    };

    Filer f = this.env.getFiler();
    PrintWriter out;
    try {
      out = f.createTextFile(Filer.Location.SOURCE_TREE, "", new File("todo.txt"), null);

      // Recherche des entités annotées avec Todo
      Collection<TypeDeclaration> types = annFilter.filter(env.getSpecifiedTypeDeclarations());
      for (TypeDeclaration typeDecl : types) {
        out.println("class name: " + typeDecl.getSimpleName());

        Todo todo = typeDecl.getAnnotation(Todo.class);

        out.println("description : ");
        for (String desc : todo.description()) {
          out.println(desc);
        }
        out.println("");
      }

      out.close();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat de l'exécution

```
C:\Documents and Settings\jm\workspace\Tests>apt -cp ".;./bin;C:/Program Files/
Java/jdk1.5.0_07/lib/tools.jar" -factory com.jmdoudoux.test.annotations.outils.T
odoAnnotationProcessorFactory com/jmdoudoux/test/*.java

C:\Documents and Settings\jm\workspace\Tests>dir
 Volume in drive C has no label.
 Volume Serial Number is 1D31-4F67

 Directory of C:\Documents and Settings\jmd\workspace\Tests

19/11/2007  08:39    <DIR>          .
19/11/2007  08:39    <DIR>          ..
16/11/2007  08:15               433 .classpath
31/10/2006  14:06               381 .project
14/09/2007  12:45    <DIR>          .settings
16/11/2007  08:15    <DIR>          bin
02/10/2007  15:22               854 build.xml
29/06/2007  07:12    <DIR>          com
15/11/2007  13:01    <DIR>          doc
19/11/2007  08:39               148 todo.txt
              8 File(s)          1 812 bytes
               6 Dir(s)  66 885 595 136 bytes free

C:\Documents and Settings\jm\workspace\Tests>type todo.txt
class name: MaClasse
description :
Corriger le bug dans le calcul

class name: MaClasse1
description :
Ajouter le traitement des erreurs
```

L'API mirror fourni de nombreuses autres fonctionnalités concernant les entités à traiter, le parcours des sources via des classes mettant en oeuvre de motif de conception visiteur, ... qui permettent de rendre très riche le traitement des annotations.

10.6.3. L'exploitation des annotations par introspection

Pour qu'une annotation soit exploitée à l'exécution, il est nécessaire qu'elle soit annotée avec une RetentionPolicy à la valeur RUNTIME.

Exemple :

```
package com.jmdoudoux.test.annotations;

import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Documented
@Retention(RetentionPolicy.RUNTIME)
public @interface Todo {

  public enum Importance {
    MINEURE, IMPORTANT, MAJEUR, CRITIQUE
  };

  Importance importance() default Importance.MINEURE;

  String[] description();

  String assigneA();

  String dateAssignation();
}
```

L'interface java.lang.reflect.AnnotatedElement définit les méthodes pour le traitement des annotations par introspection :

Méthode	Rôle
<T extends Annotation> getAnnotation(Class<T>)	Renvoyer l'annotation si le type fourni en paramètre est utilisé sur l'entité, sinon null
Annotation[] getAnnotations()	Renvoyer un tableau de toutes les annotations de l'entité. Renvoie un tableau vide si aucune annotation n'est concernée
Annotation[] getDeclaredAnnotations()	Renvoyer un tableau des annotations directement associées à l'entité (en ignorant donc les annotations héritées). Renvoie un tableau vide si aucune annotation n'est concernée
boolean isAnnotationPresent(Class< ? extends Annotation>)	Renvoyer true si l'annotation dont le type est fourni en paramètre est utilisé sur l'entité. Cette méthode est particulièrement utile dans le traitement des annotations de type marqueur.

Plusieurs classes du package java.lang implémentent l'interface AnnotatedElement : AccessibleObject, Class, Constructor, Field, Method et Package

Exemple :

```
package com.jmdoudoux.test;

import java.lang.reflect.Method;

import com.jmdoudoux.test.annotations.Todo;
import com.jmdoudoux.test.annotations.Todo.Importance;

@Todo(importance = Importance.CRITIQUE,
      description = "Corriger le bug dans le calcul",
      assigneA = "JMD",
      dateAssignation = "11-11-2007")
public class TestInstrospectionAnnotation {

  public static void main(
      String[] args) {
    Todo todo = null;

    // traitement annotation sur la classe
    Class classe = TestInstrospectionAnnotation.class;
    todo = (Todo) classe.getAnnotation(Todo.class);
    if (todo != null) {
      System.out.println("classe "+classe.getName());
      System.out.println("  ["+todo.importance()+"]"+" ("+todo.assigneA()
            +" le "+todo.dateAssignation()+")");
      for(String desc : todo.description()) {
        System.out.println("     _ "+desc);
      }
    }

    // traitement annotation sur les méthodes de la classe
    for(Method m : TestInstrospectionAnnotation.class.getMethods()) {
      todo = (Todo) m.getAnnotation(Todo.class);
      if (todo != null) {
        System.out.println("methode "+m.getName());
        System.out.println("  ["+todo.importance()+"]"+" ("+todo.assigneA()
                  +" le "+todo.dateAssignation()+")");
        for(String desc : todo.description()) {
          System.out.println("     _ "+desc);
        }
      }
    }
  }

  @Todo(importance = Importance.MAJEUR,
        description = "Implementer la methode",
        assigneA = "JMD",
        dateAssignation = "11-11-2007")
```

```
  public void methode1() {

  }

  @Todo(importance = Importance.MINEURE,
        description = {"Completer la methode", "Ameliorer les logs"},
        assigneA = "JMD",
        dateAssignation = "12-11-2007")
  public void methode2() {

  }

}
```

Résultat d'exécution :

```
classe com.jmdoudoux.test.TestInstrospectionAnnotation
  [CRITIQUE] (JMD le 11-11-2007)
     _ Corriger le bug dans le calcul
methode methode1
  [MAJEUR] (JMD le 11-11-2007)
     _ Implementer la methode
methode methode2
  [MINEURE] (JMD le 12-11-2007)
     _ Completer la methode
     _ Ameliorer les logs
```

Pour obtenir les annotations sur les paramètres d'un constructeur ou d'une méthode, il faut utiliser la méthode getParameterAnnotations() des classes Constructor ou Method qui renvoie un objet de type Annotation[][]. La première dimension du tableau concerne les paramètres dans leur ordre de déclaration. La seconde dimension contient les annotations de chaque paramètre.

10.6.4. L'exploitation des annotations par le compilateur Java

Dans la version 6 de Java SE, la prise en compte des annotations est intégrée dans le compilateur : ceci permet un traitement à la compilation des annotations sans avoir un recours à un outil tiers comme apt.

Une nouvelle API a été défini par la JSR 269 (Pluggable annotations processing API) et ajoutée dans la package javax.annotation.processing.

Cette API est détaillée dans la section suivante.

10.7. L'API Pluggable Annotation Processing

La version 6 de Java apporte plusieurs améliorations dans le traitement des annotations notamment l'intégration de ces traitements directement dans le compilateur javac grâce à une nouvelle API dédiée.

L'API Pluggable Annotation Processing est définie dans la JSR 269. Elle permet un traitement des annotations directement par le compilateur en proposant une API aux développeurs pour traiter les annotations incluses dans le code source.

Apt et son API proposaient déjà une solution à ces traitements mais cette API standardise le traitement des annotations au moment de la compilation. Il n'est donc plus nécessaire d'utiliser un outil tiers post compilation pour traiter les annotations à la compilation.

Dans les exemples de cette section, les classes suivantes seront utilisées

Exemple : MaClasse.java

```
package com.jmdoudoux.tests;

import com.jmdoudoux.tests.annotations.Todo;
import com.jmdoudoux.tests.annotations.Todo.Importance;

@Todo(importance = Importance.CRITIQUE,
      description = "Corriger le bug dans le calcul",
      assigneA = "JMD",
      dateAssignation = "11-11-2007")
public class MaClasse {

}
```

Exemple : MaClasse1.java

```
package com.jmdoudoux.tests;

import com.jmdoudoux.tests.annotations.Todo;
import com.jmdoudoux.tests.annotations.Todo.Importance;

@Todo(importance = Importance.MAJEUR,
      description = "Ajouter le traitement des erreurs",
      assigneA = "JMD",
      dateAssignation = "07-11-2007")
public class MaClasse1 {

}
```

Exemple : MaClasse3.java

```
package com.jmdoudoux.tests;

@Deprecated
public class MaClasse3 {

}
```

Un exemple de mise en oeuvre de l'API est aussi fourni avec le JDK dans le sous répertoire sample/javac/processing

10.7.1. Les processeurs d'annotations

La mise en oeuvre de cette API nécessite l'utilisation des packages javax.annotation.processing, javax.lang.model et javax.tools.

Un processeur d'annotations doit implémenter l'interface Processor. Le traitement des annotations se fait en plusieurs passes (round). A chaque passe le processeur est appelé pour traiter des classes qui peuvent avoir été générées lors de la précédente passe. Lors de la première passe, ce sont les classes fournies initialement qui sont traitées.

L'interface javax.annotation.processing Processor définit les méthodes d'un processeur d'annotations. Pour définir un processeur, il est possible de créer une classe qui implémente l'interface Processor mais le plus simple est d'hériter de la classe abstraite javax.annotation.processing.AbstractProcessor.

La classe AbstractProcessor contient une variable nommée processingEnv de type ProcessingEnvironment. La classe ProcessingEnvironment permet d'obtenir des instances de classes qui permettent des traitements avec l'extérieure du processeur ou fournissent des utilitaires :

- Filer : classe qui permet la création de fichiers
- Messager : classe qui permet d'envoyer des messages affichés par le compilateur
- Elements : classe qui fournit des utilitaires pour les éléments
- Types : classe qui fournit des utilitaires pour les types

La méthode getRootElements() renvoie les classes Java qui seront traitées par le processeur dans cette passe.

La méthode la plus importante est la méthode process() : c'est elle qui va contenir les traitements exécutés par le processeur. Elle possède deux paramètres :

- Un ensemble des annotations qui seront traitées par le processeur
- Un objet qui encapsule l'étape courante des traitements

Deux annotations sont dédiées aux processeurs d'annotations et doivent être utilisées sur la classe du processeur :

- @SupportedAnnotationTypes : cette annotation permet de préciser quelles seront les types annotations traitées par le processeur. La valeur « * » permet d'indiquer que tous les types seront traités.
- @SupportedSourceVersion : cette annotation permet de préciser la version du code source traité par le processeur

Exemple :

```
package com.jmdoudoux.tests.annotations.outils;

import java.util.Set;

import javax.annotation.processing.AbstractProcessor;
import javax.annotation.processing.Messager;
import javax.annotation.processing.RoundEnvironment;
import javax.annotation.processing.SupportedAnnotationTypes;
import javax.annotation.processing.SupportedSourceVersion;
import javax.lang.model.SourceVersion;
import javax.lang.model.element.Element;
import javax.lang.model.element.TypeElement;
import javax.tools.Diagnostic.Kind;

import com.jmdoudoux.tests.annotations.Todo;

@SupportedAnnotationTypes(value = { "*" })
@SupportedSourceVersion(SourceVersion.RELEASE_6)
public class TodoProcessor extends AbstractProcessor {

  @Override
  public boolean process(
      Set<? extends TypeElement> annotations,
      RoundEnvironment roundEnv) {

    Messager messager = processingEnv.getMessager();

    for (TypeElement te : annotations) {
      messager.printMessage(Kind.NOTE, "Traitement annotation "
            + te.getQualifiedName());

      for (Element element : roundEnv.getElementsAnnotatedWith(te)) {
        messager.printMessage(Kind.NOTE, "  Traitement element "
                  + element.getSimpleName());
        Todo todo = element.getAnnotation(Todo.class);

        if (todo != null) {
          messager.printMessage(Kind.NOTE, "  affecte le " + todo.dateAssignation()
                    + " a " + todo.assigneA());
        }
      }
    }

    return true;
  }
}
```

10.7.2. L'utilisation des processeurs par le compilateur

Le compilateur javac est enrichi avec plusieurs options concernant le traitement des annotations :

Option	Rôle
-processor	permet de préciser le nom pleinement qualifié du processeur à utiliser
-proc	
-processorpath	classpath des processeurs d'annotations
-A	permet de passer des options aux processeurs d'annotations sous la forme de paire cle=valeur
-XprintRounds	option non standard qui permet d'afficher des informations sur le traitement des annotations par les processeurs
-XprintProcessorInfo	option non standard qui affiche la liste des annotations qui seront traitées par les processeurs d'annotation

Le compilateur fait appel à la méthode process() du processeur en lui passant en paramètre l'ensemble des annotations trouvées par le compilateur dans le code source.

Résultat :

```
C:\Documents and Settings\jm\workspace\TestAnnotations>javac -cp ".;./bin;C:/Pr
ogram Files/Java/jdk1.6.0/lib/tools.jar" -processor com.jmdoudoux.tests.annotati
ons.outils.TodoProcessor com/jmdoudoux/tests/*.java
Note: Traitement annotation com.jmdoudoux.tests.annotations.Todo
Note:   Traitement element MaClasse
Note:   affecte le 11-11-2007 a JMD
Note:   Traitement element MaClasse1
Note:   affecte le 07-11-2007 a JMD
Note: Traitement annotation java.lang.Deprecated
Note:   Traitement element MaClasse3
```

10.7.3. La création de nouveaux fichiers

La classe Filer permet de créer des fichiers lors du traitement des annotations.

Exemple :

```
package com.jmdoudoux.tests.annotations.outils;

import java.io.IOException;
import java.io.PrintWriter;
import java.util.Set;

import javax.annotation.processing.AbstractProcessor;
import javax.annotation.processing.Filer;
import javax.annotation.processing.Messager;
import javax.annotation.processing.RoundEnvironment;
import javax.annotation.processing.SupportedAnnotationTypes;
import javax.annotation.processing.SupportedSourceVersion;
import javax.lang.model.SourceVersion;
import javax.lang.model.element.Element;
import javax.lang.model.element.TypeElement;
import javax.lang.model.util.Elements;
import javax.tools.StandardLocation;
import javax.tools.Diagnostic.Kind;

import com.jmdoudoux.tests.annotations.Todo;

@SupportedAnnotationTypes(value = { "*" })
@SupportedSourceVersion(SourceVersion.RELEASE_6)
```

```
public class TodoProcessor2 extends AbstractProcessor {

  @Override
  public boolean process(
      Set<? extends TypeElement> annotations,
      RoundEnvironment roundEnv) {

    Filer filer = processingEnv.getFiler();
    Messager messager = processingEnv.getMessager();
    Elements eltUtils = processingEnv.getElementUtils();
    if (!roundEnv.processingOver()) {
      TypeElement elementTodo =
            eltUtils.getTypeElement("com.jmdoudoux.tests.annotations.Todo");
      Set<? extends Element> elements = roundEnv.getElementsAnnotatedWith(elementTodo);
      if (!elements.isEmpty())
        try {
          messager.printMessage(Kind.NOTE, "Creation du fichier Todo");
          PrintWriter pw = new PrintWriter(filer.createResource(
                      StandardLocation.SOURCE_OUTPUT, "", "Todo.txt")
              .openOutputStream());
          // .createSourceFile("Todo").openOutputStream());
          pw.println("Liste des todos\n");

          for (Element element : elements) {
            pw.println("\nelement:" + element.getSimpleName());
            Todo todo = (Todo) element.getAnnotation(Todo.class);
            pw.println("  affecte le " + todo.dateAssignation()
                          + " a " + todo.assigneA());
            pw.println("  description : ");
            for (String desc : todo.description()) {
              pw.println("    " + desc);
            }
          }

          pw.close();
        } catch (IOException ioe) {
          messager.printMessage(Kind.ERROR, ioe.getMessage());
        }
      else
        messager.printMessage(Kind.NOTE, "Rien a faire");
    } else
      messager.printMessage(Kind.NOTE, "Fin des traitements");

    return true;
  }
}
```

Résultat :

```
C:\Documents and Settings\jmd\workspace\TestAnnotations>javac -cp ".;./bin;C:/Pr
ogram Files/Java/jdk1.6.0/lib/tools.jar" -processor com.jmdoudoux.tests.annotati
ons.outils.TodoProcessor2 com/jmdoudoux/tests/*.java
Note: Creation du fichier Todo
Note: Fin des traitements

C:\Documents and Settings\jmd\workspace\TestAnnotations>type Todo.txt
Liste des todos


element:MaClasse
  affecte le 11-11-2007 a JMD
  description :
    Corriger le bug dans le calcul

element:MaClasse1
  affecte le 07-11-2007 a JMD
  description :
    Ajouter le traitement des erreurs
```

10.8. Les ressources relatives aux annotations

La JSR 175 A Metadata Facility for the JavaTM Programming Language

La JSR 269 Pluggable Annotation Processing API

La JSR 250 Common Annotations

La page des annotations dans la documentation du JDK

La page des annotations dans le tutorial Java

La page d'utilisation de l'outil APT dans la documentation du JDK

Le projet open source XDoclet qui propose la génération de code à partir d'attributs dans le code

Partie 2 : Développement des interfaces graphiques

Les interfaces graphiques assurent le dialogue entre les utilisateurs et une application.

Dans un premier temps, Java propose l'API AWT pour créer des interfaces graphiques. Depuis, Java propose une nouvelle API nommée Swing. Ces deux API peuvent être utilisées pour développer des applications ou des applets. Face aux problèmes de performance de Swing, IBM a développé sa propre bibliothèque nommée SWT utilisée pour développé l'outil Eclipse. La vélocité de cette application favorise une utilisation grandissante de cette bibliothèque.

Cette partie contient les chapitres suivants :

- Le graphisme : entame une série de chapitres sur les interfaces graphiques en détaillant les objets et méthodes de base pour le graphisme
- Les éléments d'interfaces graphiques de l'AWT : recense les différents composants qui sont fournis dans la bibliothèque AWT
- La création d'interfaces graphiques avec AWT : indique comment réaliser des interfaces graphiques avec l'AWT
- L'interception des actions de l'utilisateur : détaille les mécanismes qui permettent de réagir aux actions de l'utilisateur via une interface graphique
- Le développement d'interfaces graphiques avec SWING : indique comment réaliser des interfaces graphiques avec Swing
- Le développement d'interfaces graphiques avec SWT : indique comment réaliser des interfaces graphiques avec SWT
- JFace : présente l'utilisation de ce framework facilitant le développement d'applications utilisant SWT

11. Le graphisme

La classe Graphics contient les outils nécessaires pour dessiner. Cette classe est abstraite et elle ne possède pas de constructeur public : il n'est pas possible de construire des instances de graphics nous même. Les instances nécessaires sont fournies par le système d'exploitation qui instanciera via à la machine virtuelle une sous classe de Graphics dépendante de la plateforme utilisée.

Ce chapitre contient plusieurs sections :

- Les opérations sur le contexte graphique

11.1. Les opérations sur le contexte graphique

11.1.1. Le tracé de formes géométriques

A l'exception des lignes, toutes les formes peuvent être dessinées vides (méthode drawXXX) ou pleines (fillXXX).

La classe Graphics possède de nombreuses méthodes qui permettent de réaliser des dessins.

Méthode	Rôle
drawRect(x, y, largeur, hauteur) fillRect(x, y, largeur, hauteur)	dessiner un carré ou un rectangle
drawRoundRect(x, y, largeur, hauteur, hor_arr,ver_arr) fillRoundRect(x, y, largeur, hauteur, hor_arr, ver_arr)	dessiner un carré ou un rectangle arrondi
drawLine(x1, y1, x2, y2)	Dessiner une ligne
drawOval(x, y, largeur, hauteur) fillOval(x, y, largeur, hauteur)	dessiner un cercle ou une ellipse en spécifiant le rectangle dans lequel ils s'inscrivent

drawPolygon(int[], int[], int)
fillPolygon(int[], int[], int)
	Dessiner un polygone ouvert ou fermé. Les deux premiers paramètres sont les coordonnées en abscisses et en ordonnées. Le dernier paramètre est le nombre de points du polygone.
	Pour dessiner un polygone fermé il faut joindre le dernier point au premier.
	
	Exemple (code Java 1.1) :
	`int[] x={10,60,100,80,150,10};`
	`int[] y={15,15,25,35,45,15};`
	`g.drawPolygon(x,y,x.length);`

	`g.fillPolygon(x,y,x.length);`
	
	Il est possible de définir un objet Polygon.
	
	Exemple (code Java 1.1) :
	`int[] x={10,60,100,80,150,10};`
	`int[] y={15,15,25,35,45,15};`
	`Polygon p = new Polygon(x, y,x.length );`
	`g.drawPolygon(p);`

drawArc(x, y, largeur, hauteur, angle_deb, angle_bal) fillArc(x, y, largeur, hauteur, angle_deb, angle_bal);	dessiner un arc d'ellipse inscrit dans un rectangle ou un carré. L'angle 0 se situe à 3 heures. Il faut indiquer l'angle de début et l'angle balayé

11.1.2. Le tracé de texte

La méthode drawString() permet d'afficher un texte aux coordonnées précisées

Exemple (code Java 1.1) :

```
g.drawString(texte, x, y );
```

Pour afficher des nombres de type int ou float, il suffit de les concaténer à une chaîne éventuellement vide avec l'opérateur +.

11.1.3. L'utilisation des fontes

La classe Font permet d'utiliser une police de caractères particulière pour affiche un texte.

Exemple (code Java 1.1) :

```
Font fonte = new Font(" TimesRoman ",Font.BOLD,30);
```

Le constructeur de la classe Font est Font(String, int, int). Les paramètres sont : le nom de la police, le style (BOLD, ITALIC, PLAIN ou 0,1,2) et la taille des caractères en points.

Pour associer plusieurs styles, il suffit de les additionner

Exemple (code Java 1.1) :

```
Font.BOLD + Font.ITALIC
```

Si la police spécifiée n'existe pas, Java prend la fonte par défaut même si une autre a été spécifiée précédemment. Le style et la taille seront tout de même adaptés. La méthode getName() de la classe Font retourne le nom de la fonte.

La méthode setFont() de la classe Graphics permet de changer la police d'affichage des textes

Exemple (code Java 1.1) :

```
Font fonte = new Font("
TimesRoman ",Font.BOLD,30);
g.setFont(fonte);
```

```
g.drawString("bonjour",50,50);
```

Les polices suivantes sont utilisables : Dialog, Helvetica, TimesRoman, Courier, ZapfDingBats

11.1.4. La gestion de la couleur

La méthode setColor() permet de fixer la couleur des éléments graphiques des objets de type Graphics créés après à son appel.

Exemple (code Java 1.1) :

```
g.setColor(Color.black); //(green, blue, red, white, black, ...)
```

11.1.5. Le chevauchement de figures graphiques

Si 2 surfaces de couleur différentes se superposent, alors la dernière dessinée recouvre la précédente sauf si on invoque la méthode setXORMode(). Dans ce cas, la couleur de l'intersection prend une autre couleur. L'argument à fournir est une couleur alternative. La couleur d'intersection représente une combinaison de la couleur originale et de la couleur alternative.

11.1.6. L'effacement d'une aire

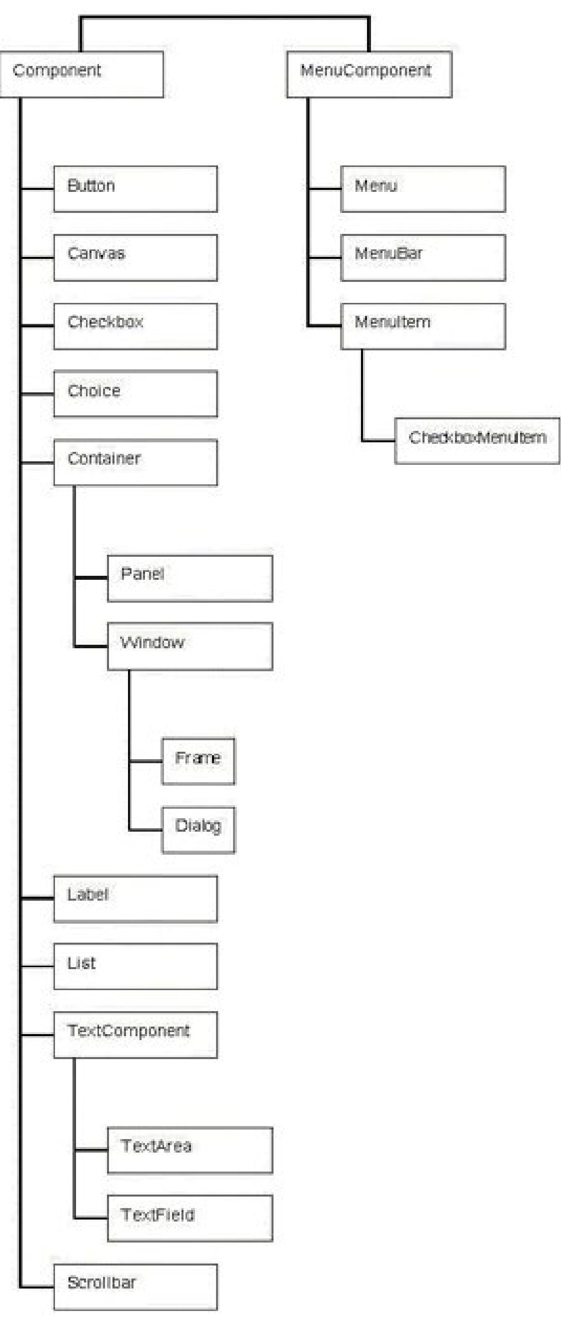
La méthode clearRect(x1, y1, x2, y2) dessine un rectangle dans la couleur de fond courante.

11.1.7. La copier une aire rectangulaire

La méthode copyArea(x1, y1, x2, y2, dx, dy) permet de copier une aire rectangulaire. Les paramètres dx et dy permettent de spécifier un décalage en pixels de la copie par rapport à l'originale.

12. Les éléments d'interfaces graphiques de l'AWT

Les classes du toolkit AWT (Abstract Windows Toolkit) permettent d'écrire des interfaces graphiques indépendantes du système d'exploitation sur lesquelles elles vont fonctionner. Cette librairie utilise le système graphique de la plateforme d'exécution (Windows, MacOS, X-Window) pour afficher les objets graphiques. Le toolkit contient des classes décrivant les composants graphiques, les polices, les couleurs et les images.



Le diagramme ci dessus définit une vue partielle de la hiérarchie des classes (les relations d'héritage) qu'il ne faut pas confondre avec la hiérarchie interne à chaque application qui définit l'imbrication des différents composants graphiques.

Les deux classes principales d'AWT sont Component et Container. Chaque type d'objet de l'interface graphique est une classe dérivée de Component. La classe Container, qui hérite de Component est capable de contenir d'autres objets graphiques (tout objet dérivant de Component).

Ce chapitre contient plusieurs sections :

- Les composants graphiques
- La classe Component
- Les conteneurs
- Les menus
- La classe java.awt.Desktop

12.1. Les composants graphiques

Pour utiliser un composant, il faut créer un nouvel objet représentant le composant et l'ajouter à un de type conteneur qui existe avec la méthode add().

Exemple (code Java 1.1) : ajout d'un bouton dans une applet (Applet hérite de Panel)

```
import java.applet.*;
import java.awt.*;

public class AppletButton extends Applet {

   Button b = new Button(" Bouton ");

   public void init() {
      super.init();
      add(b);
   }
}
```

12.1.1. Les étiquettes

Il faut utiliser un objet de la classe java.awt.Label

Exemple (code Java 1.1) :

```
Label la = new Label( );
la.setText("une etiquette");
// ou Label la = new Label("une etiquette");
```

Il est possible de créer un objet de la classe java.awt.Label en précisant l'alignement du texte

Exemple (code Java 1.1) :

```
Label la = new Label("etiquette", Label.RIGHT);
```

Le texte à afficher et l'alignement peuvent être modifiés dynamiquement lors de l'exécution :

Exemple (code Java 1.1) :

```
la.setText("nouveau texte");
la.setAlignment(Label.LEFT);
```

12.1.2. Les boutons

Il faut utiliser un objet de la classe java.awt.Button

Cette classe possède deux constructeurs :

Constructeur	Rôle
Button()	
Button(String)	Permet de préciser le libellé du bouton

Exemple (code Java 1.1) :

```
Button bouton = new Button();
bouton.setLabel("bouton");
// ou Button bouton = new Button("bouton");
```

Le libellé du bouton peut être modifié dynamiquement grâce à la méthode setLabel() :

Exemple (code Java 1.1) :

```
bouton.setLabel("nouveau libellé");
```

12.1.3. Les panneaux

Les panneaux sont des conteneurs qui permettent de rassembler des composants et de les positionner grâce à un gestionnaire de présentation. Il faut utiliser un objet de la classe java.awt.Panel.

Par défaut le gestionnaire de présentation d'un panel est de type FlowLayout.

Constructeur	Rôle
Panel()	Créer un panneau avec un gestionnaire de présentation de type FlowLayout
Panel(LayoutManager)	Créer un panneau avec le gestionnaire précisé en paramètre

Exemple (code Java 1.1) :

```
Panel p = new Panel();
```

L'ajout d'un composant au panel se fait grâce à la méthode add().

Exemple (code Java 1.1) :

```
p.add(new Button("bouton");
```

12.1.4. Les listes déroulantes (combobox)

Il faut utiliser un objet de la classe java.awt.Choice

Cette classe ne possède qu'un seul constructeur qui ne possède pas de paramètres.

Exemple (code Java 1.1) :

```
Choice maCombo = new Choice();
```

Les méthodes add() et addItem() permettent d'ajouter des éléments à la combobox.

Exemple (code Java 1.1) :

```
maCombo.addItem("element 1");
// ou maCombo.add("element 2");
```

Plusieurs méthodes permettent la gestion des sélections :

Méthodes	Rôle
void select(int);	sélectionner un élément par son indice : le premier élément correspond à l'indice 0. Une exception IllegalArgumentException est levée si l'indice ne correspond pas à un élément. Exemple (code Java 1.1) : `maCombo.select(0);`
void select(String);	sélectionner un élément par son contenu Aucune exception n'est levée si la chaîne de caractères ne correspond à aucun élément : l'élément sélectionné ne change pas. Exemple (code Java 1.1) : `maCombo.select("element 1");`

int countItems( );	déterminer le nombre d'élément de la liste. La méthode countItems() permet d'obtenir le nombre d'éléments de la combobox.
	
	Exemple (code Java 1.1) :
	`int n;`
	`n=maCombo.countItems();`
	
	JDK 1.1 il faut utiliser getItemCount() à la place
	
	Exemple (code Java 1.1) :
	`int n;`
	`n=maCombo.getItemCount();`

String getItem(int);	lire le contenu de l'élément d'indice n

	Exemple (code Java 1.1) :
	`String c = new String( );`
	`c = maCombo.getItem(n);`

String getSelectedItem();	déterminer le contenu de l'élément sélectionné
	Exemple (code Java 1.1) :
	`String s = new String( );`
	`s = maCombo.getSelectedItem();`

int getSelectedIndex( );	déterminer l'index de l'élément sélectionné
	Exemple (code Java 1.1) :
	`int n;`
	`n=maCombo.getSelectedIndex();`

12.1.5. La classe TextComponent

La classe TextComponent est la classe des mères des classes qui permettent l'édition de texte : TextArea et TextField.

Elle définit un certain nombre de méthodes dont ces classes héritent.

Méthodes	Rôle
String getSelectedText( );	Renvoie le texte sélectionné
int getSelectionStart( );	Renvoie la position de début de sélection
int getSelectionEnd( );	Renvoie la position de fin de sélection
String getText( );	Renvoie le texte contenu dans l'objet
boolean isEditable( );	Retourne un booléen indiquant si le texte est modifiable
void select(int start, int end);	Sélection des caractères situés entre start et end
void selectAll( );	Sélection de tout le texte
void setEditable(boolean b);	Autoriser ou interdire la modification du texte
void setText(String s);	Définir un nouveau texte

12.1.6. Les champs de texte

Il faut déclarer un objet de la classe java.awt.TextField

Il existe plusieurs constructeurs :

Constructeurs	Rôle
TextField();	
TextField(int);	prédétermination du nombre de caractères à saisir
TextField(String);	avec texte par défaut
TextField(String, int);	avec texte par défaut et nombre de caractères à saisir

Cette classe possède quelques méthodes utiles :

Méthodes	Rôle

String getText( )	lecture de la chaîne saisie
	Exemple (code Java 1.1) :
	`String saisie = new String();`
	`saisie = tf.getText( );`

int getColumns( )	lecture du nombre de caractères prédéfini
	Exemple (code Java 1.1) :
	`int i;`
	`i = tf.getColumns( );`

void setEchoCharacter()	pour la saisie d'un mot de passe : remplace chaque caractère saisi par celui fourni en paramètre
	Exemple (code Java 1.1) :
	`tf.setEchoCharacter('*');`
	`TextField tf = new TextField(10);`
	JDK 1.1 il faut utiliser la méthode setEchoChar()
	Exemple (code Java 1.1) :
	`tf.setEchoChar('*');`

12.1.7. Les zones de texte multilignes

Il faut déclarer un objet de la classe java.awt.TextArea

Il existe plusieurs constructeurs :

Constructeur	Rôle
TextArea()	
TextArea(int, int)	avec prédétermination du nombre de lignes et de colonnes
TextArea(String)	avec texte par défaut
TextArea(String, int, int)	avec texte par défaut et taille

Les principales méthodes sont :

Méthodes	Rôle
String getText()	lecture du contenu intégral de la zone de texte **Exemple (code Java 1.1) :** `String contenu = new String;`

	`contenu = ta.getText( );`

String getSelectedText( )	lecture de la portion de texte sélectionnée
	
	Exemple (code Java 1.1) :
	`String contenu = new String;`
	`contenu = ta.getSelectedText( );`

int getRows()	détermination du nombre de lignes
	
	Exemple (code Java 1.1) :
	`int n;`
	`n = ta.getRows( );`

int getColumns( )	détermination du nombre de colonnes
	
	Exemple (code Java 1.1) :
	`int n;`
	`n = ta.getColumns( );`

void insertText(String, int)	insertion de la chaîne à la position fournie
	
	Exemple (code Java 1.1) :
	`String text =`
	`new String("texte inséré");`
	`int n =10;`
	`ta.insertText(text,n);`
	
	JDK 1.1 Il faut utiliser la méthode insert()
	
	Exemple (code Java 1.1) :
	`String text = new`
	`String("texte inséré");`
	`int n =10;`
	`ta.insert(text,n);`

void setEditable(boolean)	Autoriser la modification
	
	Exemple (code Java 1.1) :
	`ta.setEditable(False);`
	`//texte non modifiable`

void appendText(String)	Ajouter le texte transmis au texte existant Exemple (code Java 1.1) : `ta.appendTexte(String text);` JDK 1.1 Il faut utiliser la méthode append()

void replaceText(String, int, int)	Remplacer par text le texte entre les positions start et end **Exemple (code Java 1.1) :** `ta.replaceText(text, 10, 20);` JDK 1.1 — il faut utiliser la méthode replaceRange()

12.1.8. Les listes

Il faut déclarer un objet de la classe java.awt.List.

Il existe plusieurs constructeurs :

Constructeur	Rôle
List( )	
List(int)	Permet de préciser le nombre de lignes affichées
List(int, boolean)	Permet de préciser le nombre de lignes affichées et l'indicateur de sélection multiple

Les principales méthodes sont :

Méthodes	Rôle

void addItem(String)	ajouter un élément
	Exemple (code Java 1.1) :
	`li.addItem("nouvel element");`
	`// ajout en fin de liste`
	JDK 1.1 — il faut utiliser la méthode add()

void addItem(String, int)	insérer un élément à un certain emplacement : le premier element est en position 0 **Exemple (code Java 1.1) :** `li.addItem("ajout ligne",2);` JDK 1.1 — il faut utiliser la méthode add()

void delItem(int)	retirer un élément de la liste
	Exemple (code Java 1.1) :
	`li.delItem(0);`
	`// supprime le premier element`

	il faut utiliser la méthode remove()
void delItems(int, int)	supprimer plusieurs éléments consécutifs entre les deux indices Exemple (code Java 1.1) : `li.delItems(1, 3);` cette méthode est deprecated
void clear()	effacement complet du contenu de la liste Exemple (code Java 1.1) : `li.clear( );` il faut utiliser la méthode removeAll()

void replaceItem(String, int)	remplacer un élément
	Exemple (code Java 1.1) :
	`li.replaceItem(`
	`"ligne remplacee", 1);`

int countItems()	nombre d'élément de la liste
	Exemple (code Java 1.1) :
	`int n;`
	`n = li.countItems( );`
	il faut utiliser la méthode getItemCount()

int getRows()	nombre de ligne de la liste
	Exemple (code Java 1.1) :
	`int n;`
	`n = li.getRows( );`

String getItem(int)	contenu d'un élément
	Exemple (code Java 1.1) :
	`String text = new String( );`
	`text = li.getItem(1);`

void select(int)	sélectionner un élément Exemple (code Java 1.1) :

	`li.select(0);`
setMultipleSelections(boolean)	déterminer si la sélection multiple est autorisée Exemple (code Java 1.1) : `li.setMultipleSelections(true);` JDK 1.1 il faut utiliser la méthode setMultipleMode()
void deselect(int)	désélectionner un élément Exemple (code Java 1.1) : `li.deselect(0);`

int getSelectedIndex( )	déterminer l'élément sélectionné en cas de sélection simple : renvoie l'indice ou -1 si aucun élément n'est sélectionné
	
	Exemple (code Java 1.1) :
	`int i;`
	`i = li.getSelectedIndex();`

int[] getSelectedIndexes( )	déterminer les éléments sélectionnés en cas de sélection multiple Exemple (code Java 1.1) : `int i[]=li.getSelectedIndexes();`

String getSelectedItem( )	déterminer le contenu en cas de sélection simple : renvoie le texte ou null si pas de sélection
	
	Exemple (code Java 1.1) :
	`String texte = new String( );`
	`texte = li.getSelectedItem( );`

String[] getSelectedItems()	déterminer les contenus des éléments sélectionnés en cas de sélection multiple : renvoie les textes sélectionnés ou null si pas de sélection
	
	Exemple (code Java 1.1) :
	`String texte[ ] =`
	`li.getSelectedItems();`
	`for (i = 0 ; i <`
	`texte.length(); i++)`
	`  System.out.println(texte[i]);`

boolean isSelected(int)	déterminer si un élément est sélectionné
	
	Exemple (code Java 1.1) :
	`boolean selection;`
	`selection = li.isSelected(0);`

	JDK 1.1 il faut utiliser la méthode isIndexSelect()
int getVisibleIndex()	renvoie l'index de l'entrée en haut de la liste Exemple (code Java 1.1) : `int top = li.getVisibleIndex();`
void makeVisible(int)	assure que l'élément précisé sera visible Exemple (code Java 1.1) : `li.makeVisible(10);`

Exemple (code Java 1.1) :

```
import java.awt.*;

class TestList {

  static public void main (String arg [ ]) {

    Frame frame = new Frame("Une liste");

    List list = new List(5,true);
    list.add("element 0");
    list.add("element 1");
    list.add("element 2");
    list.add("element 3");
    list.add("element 4");

    frame.add(List);
    frame.show();
    frame.pack();
  }
}
```

12.1.9. Les cases à cocher

Il faut déclarer un objet de la classe java.awt.Checkbox

Il existe plusieurs constructeurs :

Constructeur	Rôle
Checkbox()	
Checkbox(String)	avec une étiquette
Checkbox(String,boolean)	avec une étiquette et un état
Checkbox(String,CheckboxGroup, boolean)	avec une étiquette, dans un groupe de cases à cocher et un état

Les principales méthodes sont :

Méthodes	Rôle

void setLabel(String)	modifier l'étiquette Exemple (code Java 1.1) : `cb.setLabel( "libelle de la case : " );`
void setState(boolean)	fixer l'état Exemple (code Java 1.1) : `cb.setState( true );`

boolean getState( )	consulter l'état de la case
	
	Exemple (code Java 1.1) :
	`boolean etat;`
	`etat = cb.getState( );`

String getLabel( )	lire l'étiquette de la case
	
	Exemple (code Java 1.1) :
	`String commentaire = new String( );`
	`commentaire = cb.getLabel( );`

12.1.10. Les boutons radio

Déclarer un objet de la classe java.awt.CheckboxGroup

Exemple (code Java 1.1) :

```
CheckboxGroup rb;
Checkbox cb1 = new Checkbox(" etiquette 1 ", rb, etat1_boolean);
Checkbox cb2 = new Checkbox(" etiquette 2 ", rb, etat1_boolean);
Checkbox cb3 = new Checkbox(" etiquette 3 ", rb, etat1_boolean);
```

Les principales méthodes sont :

Méthodes	Rôle
Checkbox getCurrent()	retourne l'objet Checkbox correspondant à la réponse sélectionnée JDK 1.1 il faut utiliser la méthode getSelectedCheckbox()
void setCurrent(Checkbox)	Coche le bouton radio passé en paramètre JDK 1.1 il faut utiliser la méthode setSelectedCheckbox()

12.1.11. Les barres de défilement

Il faut déclarer un objet de la classe java.awt.Scrollbar

Il existe plusieurs constructeurs :

Constructeur	Rôle
Scrollbar( )	
Scrollbar(orientation)	
Scrollbar(orientation, valeur_initiale, visible, min, max)	

- orientation : Scrollbar.VERTICALE ou Scrollbar.HORIZONTAL
- valeur_initiale : position du curseur à la création
- visible : taille de la partie visible de la zone défilante
- min : valeur minimale associée à la barre
- max : valeur maximale associée à la barre

Les principales méthodes sont :

Méthodes	Rôle
sb.setValues(int,int,int,int)	mise à jour des paramètres de la barre Exemple (code Java 1.1) : `sb.setValues( valeur, visible, minimum, maximum );`
void setValue(int)	modifier la valeur courante Exemple (code Java 1.1) : `sb.setValue(10);`
int getMaximum( );	lecture du maximum Exemple (code Java 1.1) : `int max = sb.getMaximum( );`
int getMinimum( );	lecture du minimum Exemple (code Java 1.1) : `int min = sb.getMinimum( );`
int getOrientation( )	lecture de l'orientation Exemple (code Java 1.1) : `int o = sb.getOrientation( );`

int getValue( );	lecture de la valeur courante Exemple (code Java 1.1) : `int valeur = sb.getValue( );`
void setLineIncrement(int);	détermine la valeur à ajouter ou à ôter quand l'utilisateur clique sur une flèche de défilement JDK 1.1 — il faut utiliser la méthode setUnitIncrement()
int setPageIncrement( );	détermine la valeur à ajouter ou à ôter quand l'utilisateur clique sur le conteneur JDK 1.1 — il faut utiliser la méthode setBlockIncrement()

12.1.12. La classe Canvas

C'est un composant sans fonction particulière : il est utile pour créer des composants graphiques personnalisés.

Il est nécessaire d'étendre la classe Canvas pour en redéfinir la méthode Paint().

syntaxe : Cancas can = new Canvas( );

Exemple (code Java 1.1) :

```
import java.awt.*;

public class MonCanvas extends Canvas {

   public void paint(Graphics g) {
      g.setColor(Color.black);
      g.fillRect(10, 10, 100,50);
      g.setColor(Color.green);
      g.fillOval(40, 40, 10,10);
   }

}


import java.applet.*;
import java.awt.*;

public class AppletButton extends Applet {

   MonCanvas mc = new MonCanvas();

   public void paint(Graphics g) {
      super.paint(g);
      mc.paint(g);
   }
}
```

12.2. La classe Component

Les contrôles fenêtrés descendent plus ou moins directement de la classe AWT Component.

Cette classe contient de nombreuses méthodes :

Méthodes	Rôle
Rectangle bounds()	renvoie la position actuelle et la taille des composants JDK 1.1 utiliser la méthode getBounds().
void disable()	désactive les composants JDK 1.1 utiliser la méthode setEnabled(false).
void enable()	active les composants JDK 1.1 utiliser la méthode setEnabled(true).
void enable(boolean)	active ou désactive le composant selon la valeur du paramètre JDK 1.1 utiliser la méthode setEnabled(boolean).
Color getBackGround()	renvoie la couleur actuelle d'arrière plan
Font getFont()	renvoie la fonte utilisée pour afficher les caractères
Color getForeGround()	renvoie la couleur de premier plan
Graphics getGraphics()	renvoie le contexte graphique
Container getParent()	renvoie le conteneur (composant de niveau supérieur)
void hide()	masque l'objet JDK 1.1 utiliser la méthode setVisible().
boolean inside(int x, int y)	indique si la coordonnée écran absolue se trouve dans l'objet JDK 1.1 utiliser la méthode contains().
boolean isEnabled()	indique si l'objet est actif
boolean isShowing()	indique si l'objet est visible
boolean isVisible()	indique si l'objet est visible lorsque sont conteneur est visible
boolean isShowing()	indique si une partie de l'objet est visible
void layout()	repositionne l'objet en fonction du Layout Manager courant JDK 1.1 utiliser la méthode doLayout().
Component locate(int x, int y)	retourne le composant situé à cet endroit utiliser la méthode getComponentAt().

	JDK 1.1
Point location()	retourne l'origine du composant JDK 1.1 utiliser la méthode getLocation().
void move(int x, int y)	déplace les composants vers la position spécifiée JDK 1.1 utiliser la méthode setLocation().
void paint(Graphics);	dessine le composant
void paintAll(Graphics)	dessine le composant et ceux qui sont contenus en lui
void repaint()	redessine le composant pat appel à la méthode update()
void requestFocus();	demande le focus
void reshape(int x, inty, int w, int h)	modifie la position et la taille (unité : points écran) JDK 1.1 utiliser la méthode setBounds().
void resize(int w, int h)	modifie la taille (unité : points écran) JDK 1.1 utiliser la méthode setSize().
void setBackground(Color)	définit la couleur d'arrière plan
void setFont(Font)	définit la police
void setForeground(Color)	définit la couleur de premier plan
void show()	affiche le composant JDK 1.1 utiliser la méthode setVisible(True).
Dimension size()	détermine la taille actuelle JDK 1.1 utiliser la méthode getSize().

12.3. Les conteneurs

Les conteneurs sont des objets graphiques qui peuvent contenir d'autres objets graphiques, incluant éventuellement des conteneurs. Ils héritent de la classe Container.

Un composant graphique doit toujours être incorporé dans un conteneur :

Conteneur	Rôle
Panel	conteneur sans fenêtre propre. Utile pour ordonner les contrôles
Window	fenêtre principale sans cadre ni menu. Les objets descendants de cette classe peuvent servir à implémenter des menus
Dialog (descendant de Window)	réaliser des boîtes de dialogue simples

Frame (descendant de Window)	classe de fenêtre complètement fonctionnelle
Applet (descendant de Panel)	pas de menu. Pas de boîte de dialogue sans être incorporée dans une classe Frame.

L'insertion de composant dans un conteneur se fait grâce à la méthode add(Component) de la classe Container.

Exemple (code Java 1.1) :

```
Panel p = new Panel();

Button b1 = new button(" premier ");
p.add(b1);
Button b2;
p.add(b2 = new Button (" Deuxième ");
p.add(new Button("Troisième ");
```

12.3.1. Le conteneur Panel

C'est essentiellement un objet de rangement pour d'autres composants.

La classe Panel possède deux constructeurs :

Constructeur	Rôle
Panel()	
Panel(LayoutManager)	Permet de préciser un layout manager

Exemple (code Java 1.1) :

```
Panel p = new Panel( );

Button b = new Button(" bouton ");
p.add( b);
```

12.3.2. Le conteneur Window

La classe Window contient plusieurs méthodes dont voici les plus utiles :

Méthodes	Rôle
void pack()	Calculer la taille et la position de tous les contrôles de la fenêtre. La méthode pack() agit en étroite collaboration avec le layout manager et permet à chaque contrôle de garder, dans un premier temps sa taille optimale. Une fois que tous les contrôles ont leur taille optimale, pack() utilise ces informations pour positionner les contrôles. pack() calcule ensuite la taille de la fenêtre. L'appel à pack() doit se faire à l'intérieur du constructeur de fenêtre après insertion de tous les contrôles.
void show()	Afficher la fenêtre
void dispose()	Libérer les ressources allouées à la fenêtre

12.3.3. Le conteneur Frame

Ce conteneur permet de créer des fenêtres d'encadrement. Il hérite de la classe Window qui ne s'occupe que de l'ouverture de la fenêtre. Window ne connait pas les menus ni les bordures qui sont gérés par la classe Frame. Dans une applet, elle n'apparait pas dans le navigateur mais comme une fenêtre indépendante.

Il existe deux constructeurs :

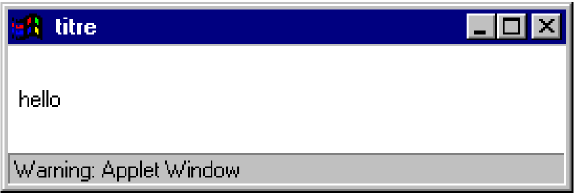
Constructeur	Rôle
Frame()	Exemple : Frame f = new Frame( );
Frame(String)	Précise le nom de la fenêtre Exemple : Frame f = new Frame(« titre »);

Les principales méthodes sont :

Méthodes	Rôle
setCursor(int)	changer le pointeur de la souris dans la fenêtre Exemple : f.setCursor(Frame.CROSSHAIR_CURSOR); JDK 1.1 utiliser la méthode setCursor(Cursor).
int getCursorType()	déterminer la forme actuelle du curseur JDK 1.1 utiliser la méthode getCursor().
Image getIconImage()	déterminer l'icone actuelle de la fenêtre
MenuBar getMenuBar()	déterminer la barre de menus actuelle
String getTitle()	déterminer le titre de la fenêtre
boolean isResizeable()	déterminer si la taille est modifiable
void remove(MenuComponent)	Supprimer un menu
void setIconImage(Image);	définir l'icone de la fenêtre
void setMenuBar(MenuBar)	Définir la barre de menu
void setResizeable(boolean)	définir si la taille peut être modifiée
void SetTitle(String)	définir le titre de la fenêtre

Exemple (code Java 1.1) :

```
import java.applet.*;
import java.awt.*;

public class AppletFrame extends Applet {

   Frame f;

   public void init() {
      super.init();
      // insert code to initialize the applet here
      f = new Frame("titre");
      f.add(new Label("hello "));
      f.show();
      f.setSize(300, 100);
   }
}
```



Le message « Warning : Applet window » est impossible à enlever dans la fenêtre : cela permet d'éviter la création d'une applet qui demande un mot de passe.

Le gestionnaire de mise en page par défaut d'une Frame est BorderLayout (FlowLayout pour une applet).

Exemple (code Java 1.1) : construction d'une fenêtre simple

```
import java.awt.*;

public class MaFrame extends Frame {

   public MaFrame() {
      super();
      setTitle(" Titre de la Fenetre ");
      setSize(300, 150);
      show(); // affiche la fenetre
   }

   public static void main(String[] args) {
      new MaFrame();
   }
}
```

12.3.4. Le conteneur Dialog

La classe Dialog hérite de la classe Window.

Une boîte de dialogue doit dérivée de la Classe Dialog de package java.awt.

Un objet de la classe Dialog doit dépendre d'un objet de la classe Frame.

Exemple (code Java 1.1) :

```
import java.awt.*;
import java.awt.event.*;

public class Apropos extends Dialog {

   public APropos(Frame parent) {
      super(parent, "A propos ", true);
      addWindowListener(new
      AProposListener(this));
      setSize(300, 300);
      setResizable(False);
   }
}

class AProposListener extends WindowAdapter {

   Dialog dialogue;
   public AProposListener(Dialog dialogue) {
      this.dialogue = dialog;
   }

   public void windowClosing(WindowEvent e) {
      dialogue.dispose();
   }
```

```
}
```

L'appel du constructeur Dialog(Frame, String, Boolean) permet de créer une instance avec comme paramètres : la fenêtre à laquelle appartient la boîte de dialogue, le titre de la boîte, le caractère modale de la boîte.

La méthode dispose() de la classe Dialog ferme la boîte et libère les ressources associées. Il ne faut pas associer cette action à la méthode windowClosed() car dispose provoque l'appel de windowClosed ce qui entrainerait un appel récursif infinie.

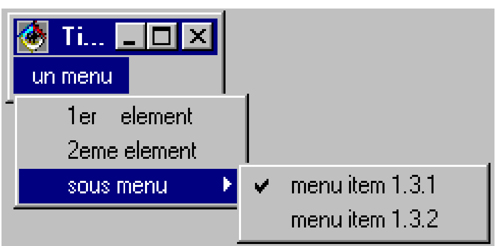
12.4. Les menus

Il faut insérer les menus dans des objets de la classe Frame (fenêtre d'encadrement). Il n'est donc pas possible d'insérer directement des menus dans une applet.

Il faut créer une barre de menu et l'affecter à la fenêtre d'encadrement. Il faut créer ensuite créer les entrées de chaque menu et les rattacher à la barre. Ajouter ensuite les éléments à chacun des menus.

Exemple (code Java 1.1) :

```
import java.awt.*;

public class MaFrame extends Frame {

   public MaFrame() {
      super();
      setTitle(" Titre de la Fenetre ");
      setSize(300, 150);

      MenuBar mb = new MenuBar();
      setMenuBar(mb);

      Menu m = new Menu(" un menu ");
      mb.add(m);
      m.add(new MenuItem(" 1er element "));
      m.add(new MenuItem(" 2eme element "));
      Menu m2 = new Menu(" sous menu ");

      CheckboxMenuItem cbm1 = new CheckboxMenuItem(" menu item 1.3.1 ");
      m2.add(cbm1);
      cbm1.setState(true);
      CheckboxMenuItem cbm2 = new CheckboxMenuItem(" menu item 1.3.2 ");
      m2.add(cbm2);

      m.add(m2);

      pack();
      show(); // affiche la fenetre

   }

   public static void main(String[] args) {
      new MaFrame();
   }

}
```



Exemple (code Java 1.1) : création d'une classe qui définit un menu

```
import java.awt.*;

public class MenuFenetre extends java.awt.MenuBar {

   public MenuItem menuQuitter, menuNouveau, menuApropos;

   public MenuFenetre() {

      Menu menuFichier = new Menu(" Fichier ");
      menuNouveau = new MenuItem(" Nouveau ");
      menuQuitter = new MenuItem(" Quitter ");

      menuFichier.add(menuNouveau);

      menuFichier.addSeparator();

      menuFichier.add(menuQuitter);

      Menu menuAide = new Menu(" Aide ");
      menuApropos = new MenuItem(" A propos ");
      menuAide.add(menuApropos);

      add(menuFichier);

      setHelpMenu(menuAide);
   }
}
```

La méthode setHelpMenu() confère sous certaines plateformes un comportement particulier à ce menu.

La méthode setMenuBar() de la classe Frame prend en paramètre une instance de la classe MenuBar. Cette instance peut être directement une instance de la classe MenuBar qui aura été modifiée grâce aux méthodes add() ou alors une classe dérivée de MenuBar qui est adaptée aux besoins (voir Exemple);

Exemple (code Java 1.1) :

```
import java.awt.*;

public class MaFrame extends Frame {

   public MaFrame() {
      super();
      setTitle(" Titre de la Fenetre ");
      setSize(300, 150);
      MenuFenetre mf = new
      MenuFenetre();

      setMenuBar(mf);

      pack();

      show(); // affiche la fenetre

   }

   public static void main(String[] args) {
      new MaFrame();
   }
```

```
}
```

12.4.1. Les méthodes de la classe MenuBar

Méthodes	Rôle
void add(Menu)	ajouter un menu dans la barre
int countMenus()	renvoie le nombre de menus JDK 1.1 utiliser la méthode getMenuCount().
Menu getMenu(int pos)	renvoie le menu à la position spécifiée
void remove(int pos)	supprimer le menu à la position spécifiée
void remove(Menu)	supprimer le menu de la barre de menu

12.4.2. Les méthodes de la classe Menu

Méthodes	Rôle
MenuItem add(MenuItem) void add(String)	ajouter une option dans le menu
void addSeparator()	ajouter un trait de séparation dans le menu
int countItems()	renvoie le nombre d'options du menu JDK 1.1 utiliser la méthode getItemCount().
MenuItem getItem(int pos)	déterminer l'option du menu à la position spécifiée
void remove(MenuItem mi)	supprimer la commande spécifiée
void remove(int pos)	supprimer la commande à la position spécifiée

12.4.3. Les méthodes de la classe MenuItem

Méthodes	Rôle
void disable()	désactiver l'élément JDK 1.1 utiliser la méthode setEnabled(false).
void enable()	activer l'élément utiliser la méthode setEnabled(true).

	JDK 1.1
void enable(boolean cond)	désactiver ou activer l'élément en fonction du paramètre JDK 1.1 utiliser la méthode setEnabled(boolean).
String getLabel()	Renvoie le texte de l'élément
boolean isEnabled()	renvoie l'état de l'élément (actif / inactif)
void setLabel(String text)	définir un nouveau texte pour la commande

12.4.4. Les méthodes de la classe CheckboxMenuItem

Méthodes	Rôle
boolean getState()	renvoie l'état d'activation de l'élément
Void setState(boolean)	définir l'état d'activation de l'élément

12.5. La classe java.awt.Desktop

Cette classe, ajoutée dans Java SE 6, permet de manipuler des documents sous la forme d'un fichier ou d'une URI à partir de leur type mime défini sur le système d'exploitation sous jacent.

La méthode statique isDesktopSupported() permet de savoir si la classe Desktop est supportée par la plate-forme.

La méthode statique Desktop.getDesktop() donne un accès à l'instance de la classe Desktop.

Plusieurs constantes sont définies dans Desktop.Action pour préciser le type d'opération qu'il est possible de réaliser sur un document : BROWSE, EDIT, MAIL, OPEN et PRINT.

La méthode isSupported() permet de savoir si l'action est supportée sur la plate-forme mais cela ne signifie pas que cette action soit supportée pour tous les types mimes enregistrés sur la plate-forme.

Plusieurs méthodes permettent d'exécuter les actions : browse(), edit(), mail(), open() et print().

Exemple : ouverture du fichier fourni en paramètre

```
package com.jmdoudoux.test.java6;

import java.awt.*;
import java.io.*;

public class TestDektop {

    public static void main(String args[]) {
        if (Desktop.isDesktopSupported()) {

            Desktop desktop = Desktop.getDesktop();
            if (args.length == 1) {
                File fichier = new File(args[0]);
                if (desktop.isSupported(Desktop.Action.OPEN)) {
                    System.out.println("Ouverture du fichier " + fichier.getName());
                    try {
                        desktop.open(fichier);
                    } catch (IOException ioe) {
                        ioe.printStackTrace();
                    }
                }
            }
```

```
        }
    }
}
```

La méthode mail() attend en paramètre une uri qui doit utiliser le protocole mailto:.

La méthode browse() attend en paramètre une uri qui utiliser un protocole reconnu par le navigateur http, https, ...

13. La création d'interfaces graphiques avec AWT

Ce chapitre contient plusieurs sections :

- Le dimensionnement des composants
- Le positionnement des composants
- La création de nouveaux composants à partir de Panel
- L'activation ou la désactivation des composants

13.1. Le dimensionnement des composants

En principe, il est automatique grâce au LayoutManager. Pour donner à un composant une taille donnée, il faut redéfinir la méthode getPreferedSize() de la classe Component.

Exemple (code Java 1.1) :

```
import java.awt.*;

public class MonBouton extends Button {

   public Dimension getPreferredSize() {
      return new Dimension(800, 250);
   }

}
```

La méthode getPreferedSize() indique la taille souhaitée mais pas celle imposée. En fonction du Layout Manager, le composant pourra ou non imposer sa taille.

Layout	Hauteur	Largeur
Sans Layout	oui	oui
FlowLayout	oui	oui
BorderLayout(East, West)	non	oui
BorderLayout(North, South)	oui	non
BorderLayout(Center)	non	non
GridLayout	non	non

Cette méthode oblige à sous classer tous les composants.

Une autre façon de faire est de se passer des Layout et de placer les composants à la main en indiquant leurs coordonnées et leurs dimensions.

Pour supprimer le Layout par défaut d'une classe, il faut appeler la méthode setLayout() avec comme paramètre null.

Trois méthodes de la classe Component permettent de positionner des composants :

- setBounds(int x, int y, int largeur, int hauteur)
- setLocation(int x , int y)
- setSize(int largeur, int hauteur)

Ces méthodes permettent de placer un composant à la position (x,y) par rapport au conteneur dans lequel il est inclus et d'indiquer sa largeur et sa hauteur.

Toutefois, les Layout Manager constituent un des facteurs importants de la portabilité des interfaces graphiques notamment en gérant la disposition et le placement des composants après redimensionnement du conteneur.

13.2. Le positionnement des composants

Lorsqu'on intègre un composant graphique dans un conteneur, il n'est pas nécessaire de préciser son emplacement car il est déterminé de façon automatique : la mise en forme est dynamique. On peut influencer cette mise en page en utilisant un gestionnaire de mise en page (Layout Manager) qui définit la position de chaque composant inséré. Dans ce cas, la position spécifiée est relative par rapport aux autres composants.

Chaque layout manager implémente l'interface java.awt.LayoutManager.

Il est possible d'utiliser plusieurs gestionnaires de mise en forme pour définir la présentation des composants. Par défaut, c'est la classe FlowLayout qui est utilisée pour la classe Panel et la classe BorderLayout pour Frame et Dialog.

Pour affecter une nouvelle mise en page, il faut utiliser la méthode setLayout() de la classe Container.

Exemple (code Java 1.1) :

```
   Panel p = new Panel();
   FlowLayout fl = new GridLayout(5,5);
   p.setLayout(fl);

   // ou p.setLayout( new GridLayout(5,5));
```

Les layout manager ont 3 avantages :

- l'aménagement des composants graphiques est délégué aux layout manager (il est inutile d'utiliser les coordonnées absolues)
- en cas de redimensionnement de la fenêtre, les contrôles sont automatiquement agrandis ou réduits
- ils permettent une indépendance vis à vis des plateformes.

Pour créer un espace entre les composants et le bord de leur conteneur, il faut redéfinir la méthode getInsets() d'un conteneur : cette méthode est héritée de la classe Container.

Exemple (code Java 1.1) :

```
public Insets getInsets() {
   Insets normal = super.getInsets();
   return new Insets(normal.top + 10, normal.left + 10,
   normal.bottom + 10, normal.right + 10);
}
```

Cet exemple permet de laisser 10 pixels en plus entre chaque bords du conteneur.

13.2.1. La mise en page par flot (FlowLayout)

La classe FlowLayout (mise en page flot) place les composants ligne par ligne de gauche à droite. Chaque ligne est complétée progressivement jusqu'à être remplie, puis passe à la suivante. Chaque ligne est centrée par défaut. C'est la mise en page par défaut des applets.

Il existe plusieurs constructeurs :

Constructeur	Rôle
FlowLayout( );	
FlowLayout(int align);	Permet de préciser l'alignement des composants dans le conteneur (CENTER, LEFT, RIGHT ...). Par défaut, align vaut CENTER
FlowLayout(int, int hgap, int vgap);	Permet de préciser et l'alignement horizontal et vertical dont la valeur par défaut est 5.

Exemple (code Java 1.1) :

```
import java.awt.*;

public class MaFrame extends Frame {

   public MaFrame() {
      super();
      setTitle(" Titre de la Fenetre ");
      setSize(300, 150);
      setLayout(new FlowLayout());
      add(new Button("Bouton 1"));
      add(new Button("Bouton 2"));
      add(new Button("Bouton 3"));

      pack();

      show(); // affiche la fenetre

   }

   public static void main(String[] args) {

      new MaFrame();

   }

}
```

Chaque applet possède une mise en page flot implicitement initialisée à FlowLayout(FloawLayout.CENTER,5,5).

FlowLayout utilise les dimensions de son conteneur comme seul principe de mise en forme des composants. Si les dimensions du conteneur changent, le positionnement des composants est recalculé.

Exemple : la fenêtre précédente est simplement redimensionnée

13.2.2. La mise en page bordure (BorderLayout)

Avec ce Layout Manager, la disposition des composants est commandée par une mise en page en bordure qui découpe la surface en cinq zones : North, South, East, West, Center. On peut librement utiliser une ou plusieurs zones.

BorderLayout consacre tout l'espace du conteneur aux composants. Le composant du milieu dispose de la place inutilisée par les autres composants.

Il existe plusieurs constructeurs :

Constructeur	Rôle
BorderLayout( )	
BorderLayout (int hgap,int vgap)	Permet de préciser l'espacement horizontal et vertical des composants.

Exemple (code Java 1.1) :

```
import java.awt.*;

public class MaFrame extends Frame {

   public MaFrame() {
      super();
      setTitle("
      Titre de la Fenetre ");
      setSize(300, 150);
      setLayout(new
      BorderLayout());
      add("North", new Button(" bouton haut "));
      add("South", new Button(" bouton bas "));
      add("West",  new Button(" bouton gauche "));
      add("East",  new Button(" bouton droite "));
      add("Center", new Button(" bouton milieu "));
      pack();
      show(); // affiche la fenetre
   }

   public static void
      main(String[] args) {
      new MaFrame();
   }
}
```

Il est possible d'utiliser deux méthodes add surchargées de la classe Container : add(String, Component) ou le premier paramètre précise l'orientation du composant ou add(Component, Objet) ou le second paramètre précise la position sous forme de constante définie dans la classe BorderLayout.

Exemple (code Java 1.1) :

```
import java.awt.*;

public class MaFrame extends Frame {

   public MaFrame() {

      super();
```

```
      setTitle(" Titre de la Fenetre ");
      setSize(300, 150);
      setLayout(new BorderLayout());
      add(new Button("North"), BorderLayout.NORTH);
      add(new Button("South"), BorderLayout.SOUTH);
      pack();
      show(); // affiche la fenetre

   }

   public static void main(String[] args) {

      new MaFrame();

   }

}
```

13.2.3. La mise en page de type carte (CardLayout)

Ce layout manager aide à construire des boîtes de dialogue composées de plusieurs onglets. Un onglet se compose généralement de plusieurs contrôles : on insère des panneaux dans la fenêtre utilisée par le CardLayout Manager. Chaque panneau correspond à un onglet de boîte de dialogue et contient plusieurs contrôles. Par défaut, c'est le premier onglet qui est affiché.

Ce layout possède deux constructeurs :

Constructeurs	Rôle
CardLayout()	
CardLayout(int, int)	Permet de préciser l'espace horizontal et vertical du tour du composant

Exemple (code Java 1.1) :

```
import java.awt.*;

public class MaFrame extends Frame {

   public MaFrame() {

      super();
      setTitle("Titre de la Fenetre ");
      setSize(300,150);
      CardLayout cl = new CardLayout();
      setLayout(cl);

      //création d'un panneau contenant les contrôles d'un onglet
      Panel p = new Panel();

      //ajouter les composants au panel
      p.add(new Button("Bouton 1 panneau 1"));
      p.add(new Button("Bouton 2 panneau 1"));

      //inclure le panneau dans la fenetre sous le nom "Page1"
      // ce nom est utilisé par show()

      add("Page1",p);
```

```
        //déclaration et insertion de l'onglet suivant
        p = new Panel();
        p.add(new Button("Bouton 1 panneau 2"));
        add("Page2", p);

        // affiche la fenetre
        pack();
        show();

    }

    public static void main(String[] args) {
        new MaFrame();
    }

}
```

Lors de l'insertion d'un onglet, un nom doit lui être attribué. Les fonctions nécessaires pour afficher un onglet de boîte de dialogue ne sont pas fournies par les méthodes du conteneur, mais seulement par le Layout Manager. Il est nécessaire de sauvegarder temporairement le Layout Manager dans une variable ou déterminer le gestionnaire en cours par un appel à getLayout(). Pour appeler un onglet donné, il faut utiliser la méthode show() du CardLayout Manager.

Exemple (code Java 1.1) :

```
((CardLayout)getLayout()).show(this, "Page2");
```

Les méthodes first(), last(), next() et previous() servent à parcourir les onglets de boîte de dialogue :

Exemple (code Java 1.1) :

```
((CardLayout)getLayout()).first(this);
```

13.2.4. La mise en page GridLayout

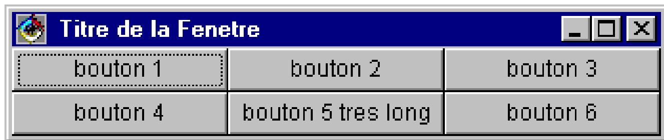
Ce Layout Manager établit un réseau de cellules identiques qui forment une sorte de quadrillage invisible : les composants sont organisés en lignes et en colonnes. Les éléments insérés dans la grille ont tous la même taille. Les cellules du quadrillage se remplissent de droite à gauche ou de haut en bas.

Il existe plusieurs constructeurs :

Constructeur	Rôle
GridLayout(int, int);	Les deux premiers entiers spécifient le nombre de lignes ou de colonnes de la grille.
GridLayout(int, int, int, int);	permet de préciser en plus l'espacement horizontal et vertical des composants.

Exemple (code Java 1.1) :

```
import java.awt.*;

public class MaFrame extends Frame {

   public MaFrame() {
      super();
      setTitle(" Titre de la Fenetre ");
      setSize(300, 150);
      setLayout(new GridLayout(2, 3));
      add(new Button("bouton 1"));
      add(new Button("bouton 2"));
      add(new Button("bouton 3"));
      add(new Button("bouton 4"));
      add(new Button("bouton 5 tres long"));
      add(new Button("bouton 6"));

      pack();

      show(); // affiche la fenetre

   }

   public static void main(String[] args) {
      new MaFrame();
   }

}
```



Attention : lorsque le nombre de ligne et de colonne est spécifié alors le nombre de colonne est ignoré. Ainsi par Exemple GridLayout(5,4) est équivalent à GridLayout(5,0).

Exemple (code Java 1.1) :

```
import java.awt.*;

public class MaFrame extends Frame {

   public MaFrame() {
      super();
      setTitle(" Titre de la Fenetre ");
      setSize(300, 150);
      setLayout(new GridLayout(2, 3));
      add(new Button("bouton 1"));
      add(new Button("bouton 2"));
      add(new Button("bouton 3"));
      add(new Button("bouton 4"));
      add(new Button("bouton 5 tres long"));
      add(new Button("bouton 6"));
      add(new Button("bouton 7"));
      add(new Button("bouton 8"));
      add(new Button("bouton 9"));

      pack();
      show(); // affiche la fenetre

   }

   public static void
      main(String[] args) {
      new MaFrame();
   }

}
```

13.2.5. La mise en page GridBagLayout

Ce gestionnaire (grille étendue) est le plus riche en fonctionnalités : le conteneur est divisé en cellules égales mais un composant peut occuper plusieurs cellules de la grille et il est possible de faire une distribution dans des cellules distinctes. Un objet de la classe GridBagConstraints permet de donner les indications de positionnement et de dimension à l'objet GridBagLayout.

Les lignes et les colonnes prennent naissance au moment ou les contrôles sont ajoutés. Chaque contrôle est associé à un objet de la classe GridBagConstraints qui indique l'emplacement voulu pour le contrôle.

Exemple (code Java 1.1) :

```
GridBagLayout gbl = new GridBagLayout( );
GridBagConstraints gbc = new GridBagConstraints( );
```

Les variables d'instances pour manipuler l'objet GridBagLayoutConstrainst sont :

Variable	Rôle
gridx et gridy	Ces variables contiennent les coordonnées de l'origine de la grille. Elles permettent un positionnement précis à une certaine position d'un composant. Par défaut elles ont la valeur GrigBagConstraint.RELATIVE qui indique qu'un composant se range à droite du précédent
gridwidth, gridheight	Définissent combien de cellules va occuper le composant (en hauteur et largeur). Par défaut la valeur est 1. L'indication est relative aux autres composants de la ligne ou de la colonne. La valeur GridBagConstraints.REMAINDER spécifie que le prochain composant inséré sera le dernier de la ligne ou de la colonne courante. La valeur GridBagConstraints.RELATIVE place le composant après le dernier composant d'une ligne ou d'une colonne.
fill	Définit le sort d'un composant plus petit que la cellule de la grille. GridBagConstraints.NONE conserve la taille d'origine : valeur par défaut GridBagConstraints.HORIZONTAL dilaté horizontalement GridBagConstraints.VERTICAL dilaté verticalement GridBagConstraints.BOTH dilatés aux dimensions de la cellule
ipadx, ipady	Permettent de définir l'agrandissement horizontal et vertical des composants. Ne fonctionne que si une dilatation est demandée par fill. La valeur par défaut est (0,0).
anchor	Lorsqu'un composant est plus petit que la cellule dans laquelle il est inséré, il peut être positionné à l'aide de cette variable pour définir le côté par lequel le contrôle doit être aligné dans la cellule. Les variables possibles sont NORTH, NORTHWEST, NORTHEAST, SOUTH, SOUTHWEST, SOUTHEAST, WEST et EAST
weightx, weighty	Permettent de définir la répartition de l'espace en cas de changement de dimension

Exemple (code Java 1.1) :

```
import java.awt.*;

public class MaFrame extends Frame {

   public MaFrame() {
      super();
      setTitle(" Titre de la Fenetre ");
      setSize(300, 150);
```

```
        Button b1 = new Button(" bouton 1 ");
        Button b2 = new Button(" bouton 2 ");
        Button b3 = new Button(" bouton 3 ");

        GridBagLayout gb = new GridBagLayout();

        GridBagConstraints gbc = new GridBagConstraints();
        setLayout(gb);

        gbc.fill = GridBagConstraints.BOTH;
        gbc.weightx = 1;
        gbc.weighty = 1;
        gb.setConstraints(b1, gbc); // mise en forme des objets
        gb.setConstraints(b2, gbc);
        gb.setConstraints(b3, gbc);

        add(b1);
        add(b2);
        add(b3);

        pack();

        show(); // affiche la fenetre

    }

    public static void main(String[] args) {
        new MaFrame();
    }

}
```

Cet exemple place trois boutons l'un à coté de l'autre. Ceci permet en cas de changement de dimension du conteneur de conserver la mise en page : la taille des composants est automatiquement ajustée.

Pour placer les 3 boutons l'un au dessus de l'autre, il faut affecter la valeur 1 à la variable gbc.gridx.

13.3. La création de nouveaux composants à partir de Panel

Il est possible de définir de nouveau composant qui hérite directement de Panel

Exemple (code Java 1.1) :

```
class PanneauClavier extends Panel {

    PanneauClavier()
    {
        setLayout(new GridLayout(4,3));

        for (int num=1; num <= 9 ; num++) add(new Button(Integer.toString(num)));
        add(new Button("*");
        add(new Button("0");
        add(new Button("# ");
```

```
   }
}
public class demo exteds Applet {
   public void init() { add(new PanneauClavier()); }
}
```

13.4. L'activation ou la désactivation des composants

L'activation ou la désactivation d'un composant se fait grâce à sa méthode setEnabled(boolean). La valeur booléenne passée en paramètres indique l'état du composant (false : interdit l'usage du composant). Cette méthode est un moyen d'interdire à un composant d'envoyer des événements utilisateurs.

14. L'interception des actions de l'utilisateur

N'importe quelle interface graphique doit interagir avec l'utilisateur et donc réagir à certains événements. Le modèle de gestion de ces événements à changer entre le JDK 1.0 et 1.1.

Ce chapitre traite de la capture des ces événements pour leur associer des traitements. Il contient plusieurs sections :

- L'interception des actions de l'utilisateur avec Java version 1.0
- L'interception des actions de l'utilisateur avec Java version 1.1

14.1. L'interception des actions de l'utilisateur avec Java version 1.0

Cette section sera développée dans une version future de ce document

14.2. L'interception des actions de l'utilisateur avec Java version 1.1

Les événements utilisateurs sont gérés par plusieurs interfaces EventListener.

Les interfaces EventListener permettent à un composant de générer des événements utilisateurs. Une classe doit contenir une interface auditeur pour chaque type de composant :

- ActionListener : clic de souris ou enfoncement de la touche Enter
- ItemListener : utilisation d'une liste ou d'une case à cocher
- MouseMotionListener : événement de souris
- WindowListener : événement de fenêtre

L'ajout d'une interface EventListener impose plusieurs ajouts dans le code :

1. importer le groupe de classe java.awt.event

Exemple (code Java 1.1) :

```
import java.awt.event.*;
```

2. la classe doit déclarer qu'elle utilisera une ou plusieurs interfaces d'écoute

Exemple (code Java 1.1) :

```
public class AppletAction extends Applet implements ActionListener{
```

Pour déclarer plusieurs interfaces, il suffit de les séparer par des virgules

Exemple (code Java 1.1) :

```
public class MonApplet extends Applet implements ActionListener, MouseListener {
```

3. Appel à la méthode addXXX() pour enregistrer l'objet qui gérera les événements XXX du composant

   Il faut configurer le composant pour qu'il possède un «écouteur» pour l'événement utilisateur concerné.

   Exemple (code Java 1.1) : création d'un bouton capable de réagir à un événements

   ```
   Button b = new Button("boutton");
   b.addActionListener(this);
   ```

   Ce code crée l'objet de la classe Button et appelle sa méthode addActionListener(). Cette méthode permet de préciser qu'elle sera la classe qui va gérer l'événement utilisateur de type ActionListener du bouton. Cette classe doit impérativement implémenter l'interface de type EventListener correspondante soit dans cette exemple ActionListener. L'instruction this indique que la classe elle même recevra et gérera l'événement utilisateur.

   L'apparition d'un événement utilisateur généré par un composant doté d'un auditeur appelle automatiquement une méthode, qui doit se trouver dans la classe référencée dans l'instruction qui lie l'auditeur au composant. Dans l'exemple, cette méthode doit être située dans la même classe parce que c'est l'objet lui même qui est spécifié avec l'instruction this. Une autre classe indépendante peut être utilisée : dans ce cas il faut préciser une instance de cette classe en temps que paramètre.
4. implémenter les méthodes déclarées dans les interfaces

   Chaque auditeur possède des méthodes différentes qui sont appelées pour traiter leurs événements. Par exemple, l'interface ActionListener envoie des événements à une classe nommée actionPerformed( ).

   Exemple (code Java 1.1) :

   ```
   public void actionPerformed(ActionEvent evt) {
      //insérer ici le code de la méthode
   };
   ```

   Pour identifier le composant qui a généré l'événement, il faut utiliser la méthode getActionCommand() de l'objet ActionEvent fourni en paramètre de la méthode :

   Exemple (code Java 1.1) :

   ```
   String composant = evt.getActionCommand();
   ```

   La méthode getActionCommand() renvoie une chaîne de caractères. Si le composant est un bouton, alors il renvoie le texte du bouton, si le composant est une zone de saisie, c'est le texte saisie qui sera renvoyé (il faut appuyer sur «Entrer» pour générer l'événement), etc ...

   La méthode getSource() renvoie l'objet qui a généré l'événement. Cette méthode est plus sure que la précédente

   Exemple (code Java 1.1) :

   ```
   Button b = new Button(" bouton ");

   ...
   ```

```
void public actionPerformed(actionEvent evt) {
   Object source = evt.getSource();

   if (source == b) // action a effectuer
}
```

La méthode getSource() peut être utilisé avec tous les événements utilisateur.

Exemple (code Java 1.1) : Exemple complet qui affiche le composant qui a généré l'événement

```
package applets;

import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class AppletAction extends Applet implements ActionListener{

   public void actionPerformed(ActionEvent evt) {
      String composant = evt.getActionCommand();
      showStatus("Action sur le composant : " + composant);
   }


   public void init() {
      super.init();

      Button b1 = new Button("boutton 1");
      b1.addActionListener(this);
      add(b1);

      Button b2 = new Button("boutton 2");
      b2.addActionListener(this);
      add(b2);

      Button b3 = new Button("boutton 3");
      b3.addActionListener(this);
      add(b3);
   }
}
```

14.2.1. L'interface ItemListener

Cette interface permet de réagir à la sélection de cases à cocher et de liste d'options. Pour qu'un composant génère des événements, il faut utiliser la méthode addItemListener().

Exemple (code Java 1.1) :

```
Checkbox cb = new Checkbox(" choix ",true);
cb.addItemListener(this);
```

Ces événements sont reçus par la méthode itemStateChanged() qui attend un objet de type ItemEvent en argument

Pour déterminer si une case à cocher est sélectionnée ou inactive, utiliser la méthode getStateChange() avec les constantes ItemEvent.SELECTED ou ItemEvent.DESELECTED.

Exemple (code Java 1.1) :

```
package applets;

import java.applet.*;
import java.awt.*;
import java.awt.event.*;
```

```
public class AppletItem extends Applet implements ItemListener{

   public void init() {
      super.init();
      Checkbox cb = new Checkbox("choix 1", true);
      cb.addItemListener(this);
      add(cb);
   }

   public void itemStateChanged(ItemEvent item) {
      int status = item.getStateChange();
      if (status == ItemEvent.SELECTED)
         showStatus("choix selectionne");
      else
         showStatus("choix non selectionne");
   }

}
```

Pour connaitre l'objet qui a généré l'événement, il faut utiliser la méthode getItem().

Pour déterminer la valeur sélectionnée dans une combo box, il faut utiliser la méthode getItem() et convertir la valeur en chaîne de caractères.

Exemple (code Java 1.1) :

```
Package applets;

import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class AppletItem extends Applet implements ItemListener{

   public void init() {
      Choice c = new Choice();
      c.add("choix 1");
      c.add("choix 2");
      c.add("choix 3");
      c.addItemListener(this);
      add(c);
}

   public void itemStateChanged(ItemEvent item) {
      Object obj = item.getItem();
      String selection = (String)obj;
      showStatus("choix : "+selection);
   }
}
```

14.2.2. L'interface TextListener

Cette interface permet de réagir aux modifications de la zone de saisie ou du texte.

La méthode addTextListener() permet à un composant de texte de générer des événements utilisateur. La méthode TextValueChanged() reçoit les événements.

Exemple (code Java 1.1) :

```
package applets;

import java.applet.*;
import java.awt.*;
import java.awt.event.*;
```

```
public class AppletText extends Applet implements TextListener{

   public void init() {
      super.init();

      TextField t = new TextField("");
      t.addTextListener(this);
      add(t);
   }

   public void textValueChanged(TextEvent txt) {
      Object source = txt.getSource();
      showStatus("saisi = "+((TextField)source).getText());
   }
}
```

14.2.3. L'interface MouseMotionListener

La méthode addMouseMotionListener() permet de gérer les événements liés à des mouvements de souris. La méthode mouseDragged() et mouseMoved() reçoivent les événements.

Exemple (code Java 1.1) :

```
package applets;

import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class AppletMotion extends Applet implements MouseMotionListener{
   private int x;
   private int y;

   public void init() {
      super.init();
      this.addMouseMotionListener(this);
   }

   public void mouseDragged(java.awt.event.MouseEvent e) {}

   public void mouseMoved(MouseEvent e) {
      x = e.getX();
      y = e.getY();
      repaint();
      showStatus("x = "+x+" ; y = "+y);
   }

   public void paint(Graphics g) {
      super.paint(g);
      g.drawString("x = "+x+" ; y = "+y,20,20);
   }
}
```

14.2.4. L'interface MouseListener

Cette interface permet de réagir aux clics de souris. Les méthodes de cette interface sont :

- public void mouseClicked(MouseEvent e);
- public void mousePressed(MouseEvent e);
- public void mouseReleased(MouseEvent e);
- public void mouseEntered(MouseEvent e);

* public void mouseExited(MouseEvent e);

Exemple (code Java 1.1) :

```
package applets;

import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class AppletMouse extends Applet implements MouseListener {
   int nbClick = 0;

   public void init() {
      super.init();
      addMouseListener(this);
   }

   public void mouseClicked(MouseEvent e) {
      nbClick++;
      repaint();
   }

   public void mouseEntered(MouseEvent e) {}

   public void mouseExited(MouseEvent e) {}

   public void mousePressed(MouseEvent e) {}

   public void mouseReleased(MouseEvent e) {}

   public void paint(Graphics g) {
      super.paint(g);
      g.drawString("Nombre de clics : "+nbClick,10,10);
   }
}
```

Une classe qui implémente cette interface doit définir ces 5 méthodes. Si toutes les méthodes ne doivent pas être utilisées, il est possible de définir une classe qui hérite de MouseAdapter. Cette classe fournit une implémentation par défaut de l'interface MouseListener.

Exemple (code Java 1.1) :

```
class gestionClics extends MouseAdapter {

   public void mousePressed(MouseEvent e) {
      //traitement
   }
}
```

Dans le cas d'une classe qui hérite d'une classe Adapter, il suffit de redéfinir la ou les méthodes qui contiendront du code pour traiter les événements concernés. Par défaut, les différentes méthodes définies dans l'Adapter ne font rien.

Cette nouvelle classe ainsi définie doit être passée en paramètre à la méthode addMouseListener() au lieu de this qui indiquait que la classe répondait elle même à l'événement.

14.2.5. L'interface WindowListener

La méthode addWindwowListener() permet à un objet Frame de générer des événements. Les méthodes de cette interface sont :

* public void windowOpened(WindowEvent e)
* public void windowClosing(WindowEvent e)
* public void windowClosed(WindowEvent e)

* public void windowIconified(WindowEvent e)
* public void windowDeinconified(WindowEvent e)
* public void windowActivated(WindowEvent e)
* public void windowDeactivated(WindowEvent e)

windowClosing() est appelée lorsque l'on clique sur la case système de fermeture de la fenêtre. windowClosed() est appelé après la fermeture de la fenêtre : cette méthode n'est utile que si la fermeture de la fenêtre n'entraine pas la fin de l'application.

Exemple (code Java 1.1) :

```
package test;

import java.awt.event.*;

class GestionnaireFenetre extends WindowAdpter {

   public void windowClosing(WindowEvent e) {
      System.exit(0);
   }
}
```

Exemple (code Java 1.1) :

```
package test;

import java.awt.*;
import java.awt.event.*;

public class TestFrame extends Frame {

   private GestionnaireFenetre gf = new GestionnaireFenetre();


   public TestFrame(String title) {
     super(title);
     addWindowListener(gf);
   }

   public static void main(java.lang.String[] args) {
      try {
         TestFrame tf = new TestFrame("TestFrame");
         tf.setVisible(true);
      } catch (Throwable e) {
         System.err.println("Erreur");
         e.printStackTrace(System.out);
      }
   }
}
```

14.2.6. Les différentes implémentations des Listeners

La mise en oeuvre des Listeners peut se faire selon différentes formes : la classe implémentant elle même l'interface, une classe indépendante, une classe interne, une classe interne anonyme.

14.2.6.1. Une classe implémentant elle même le listener

Exemple (code Java 1.1) :

```
package test;

import java.awt.*;
import java.awt.event.*;
```

```
public class TestFrame3 extends Frame implements WindowListener {

   public TestFrame3(String title) {
      super(title);
      this.addWindowListener(this);
   }

   public static void main(java.lang.String[] args) {
      try {
         TestFrame3 tf = new TestFrame3("testFrame3");
         tf.setVisible(true);
      } catch (Throwable e) {
         System.err.println("Erreur");
         e.printStackTrace(System.out);
      }
   }

   public void windowActivated(java.awt.event.WindowEvent e) {}

   public void windowClosed(java.awt.event.WindowEvent e) {}

   public void windowClosing(java.awt.event.WindowEvent e) {
      System.exit(0);
   }

   public void windowDeactivated(java.awt.event.WindowEvent e) {}

   public void windowDeiconified(java.awt.event.WindowEvent e) {}

   public void windowIconified(java.awt.event.WindowEvent e) {}

   public void windowOpened(java.awt.event.WindowEvent e) {}

}
```

14.2.6.2. Une classe indépendante implémentant le listener

Exemple (code Java 1.1) :

```
package test;

import java.awt.*;
import java.awt.event.*;

public class TestFrame4 extends Frame {

   public TestFrame4(String title) {
      super(title);
      gestEvt ge = new gestEvt();
      addWindowListener(ge);
   }

   public static void main(java.lang.String[] args) {
      try {
         TestFrame4 tf = new TestFrame4("testFrame4");
         tf.setVisible(true);
      } catch (Throwable e) {
         System.err.println("Erreur");
         e.printStackTrace(System.out);
      }
   }
}
```

Exemple (code Java 1.1) :

```
package test;
```

```
import java.awt.event.*;

public class gestEvt implements WindowListener {

   public void windowActivated(WindowEvent e) {}
   public void windowClosed(WindowEvent e) {}
   public void windowClosing(WindowEvent e) {
      System.exit(0);
   }
   public void windowDeactivated(WindowEvent e) {}
   public void windowDeiconified(WindowEvent e) {}
   public void windowIconified(WindowEvent e) {}
   public void windowOpened(WindowEvent e) {}
}
```

14.2.6.3. Une classe interne

Exemple (code Java 1.1) :

```
package test;

import java.awt.*;
import java.awt.event.*;

public class TestFrame2 extends Frame {

   class gestEvt implements WindowListener {
      public void windowActivated(WindowEvent e) {};
      public void windowClosed(WindowEvent e) {};
      public void windowClosing(WindowEvent e) {
         System.exit(0);
      };
      public void windowDeactivated(WindowEvent e) {};
      public void windowDeiconified(WindowEvent e) {};
      public void windowIconified(WindowEvent e) {};
      public void windowOpened(WindowEvent e) {};
   };

   private gestEvt ge = new TestFrame2.gestEvt();


   public TestFrame2(String title) {
     super(title);
     addWindowListener(ge);
   }

   public static void main(java.lang.String[] args) {
      try {
         TestFrame2 tf = new TestFrame2("TestFrame2");
         tf.setVisible(true);
      } catch (Throwable e) {
         System.err.println("Erreur");
         e.printStackTrace(System.out);
      }
   }
}
```

14.2.6.4. Une classe interne anonyme

Exemple (code Java 1.1) :

```
package test;

import java.awt.*;
```

```
import java.awt.event.*;

public class TestFrame1 extends Frame {

   public TestFrame1(String title) {
      super(title);
      addWindowListener(new WindowAdapter() {
         public void windowClosed(.WindowEvent e) {
            System.exit(0);
         };
      });
   }

   public static void main(java.lang.String[] args) {
      try {
         TestFrame1 tf = new TestFrame1("TestFrame");
         tf.setVisible(true);
      } catch (Throwable e) {
         System.err.println("Erreur");
         e.printStackTrace(System.out);
      }
   }
}
```

14.2.7. Résumé

Le mécanisme mis en place pour intercepter des événements est le même quel que soit ces événements :

- associer au composant qui est à l'origine de l'événement un contrôleur adéquat : utilisation des méthodes addXXXListener() Le paramètre de ces méthodes indique l'objet qui a la charge de répondre au message : cet objet doit implémenter l'interface XXXListener correspondant ou dérivé d'une classe XXXAdapter (créer une classe qui implémente l'interface associé à l'événement que l'on veut gérer. Cette classe peut être celle du composant qui est à l'origine de l'événement (facilité d'implémentation) ou une classe indépendante qui détermine la frontière entre l'interface graphique (émission d'événement) et celle qui représente la logique de l'application (traitement des événements)).
- les classes XXXAdapter sont utiles pour créer des classes dédiées au traitement des événements car elles implémentent des méthodes par défaut pour celles définies dans l'interface XXXListener dérivées de EventListener. Il n'existe une classe Adapter que pour les interfaces qui possèdent plusieurs méthodes.
- implémenter la méthode associé à l'événement qui fournit en paramètre un objet de type AWTEvent (classe mère de tout événement) qui contient des informations utiles (position du curseur, état du clavier ...).

15. Le développement d'interfaces graphiques avec SWING

Swing fait partie de la bibliothèque Java Foundation Classes (JFC). C'est une API dont le but est similaire à celui de l'API AWT mais dont le mode de fonctionnement et d'utilisation est complètement différent. Swing a été intégré au JDK depuis sa version 1.2. Cette bibliothèque existe séparément. pour le JDK 1.1.

La bibliothèque JFC contient :

- l'API Swing : de nouvelles classes et interfaces pour construire des interfaces graphiques
- Accessibility API :
- 2D API: support du graphisme en 2D
- API pour l'impression et le cliquer/glisser

Ce chapitre contient plusieurs sections :

- La présentation de Swing
- Les packages Swing
- Un exemple de fenêtre autonome
- Les composants Swing
- Les boutons
- Les composants de saisie de texte
- Les onglets
- Le composant JTree
- Les menus
- L'affichage d'une image dans une application.

15.1. La présentation de Swing

Swing propose de nombreux composants dont certains possèdent des fonctions étendues, une utilisation des mécanismes de gestion d'événements performants (ceux introduits par le JDK 1.1) et une apparence modifiable à la volée (une interface graphique qui emploie le style du système d'exploitation Windows ou Motif ou un nouveau style spécifique à Java nommé Metal).

Tous les éléments de Swing font partie d'un package qui a changé plusieurs fois de nom : le nom du package dépend de la version du J.D.K. utilisée :

- com.sun.java.swing : jusqu'à la version 1.1 beta 2 de Swing, de la version 1.1 des JFC et de la version 1.2 beta 4 du J.D.K.
- java.awt.swing : utilisé par le J.D.K. 1.2 beta 2 et 3
- javax.swing : à partir des versions de Swing 1.1 beta 3 et J.D.K. 1.2 RC1

Les composants Swing forment un nouvelle hiérarchie parallèle à celle de l'AWT. L'ancêtre de cette hiérarchie est le composant JComponent. Presque tous ces composants sont écrits en pur Java : ils ne possèdent aucune partie native sauf ceux qui assurent l'interface avec le système d'exploitation : JApplet, JDialog, JFrame, et JWindow. Cela permet aux composants de toujours avoir la même apparence quelque soit le système sur lequel l'application s'exécute.

Tous les composants Swing possèdent les caractéristiques suivantes :

- ce sont des beans

* ce sont des composants légers (pas de partie native) hormis quelques exceptions.
* leurs bords peuvent être changés

La procédure à suivre pour utiliser un composant Swing est identique à celle des composants de la bibliothèque AWT : créer le composant en appelant son constructeur, appeler les méthodes du composant si nécessaire pour le personnaliser et l'ajouter dans un conteneur.

Swing utilise la même infrastructure de classes qu'AWT, ce qui permet de mélanger des composants Swing et AWT dans la même interface. Sun recommande toutefois d'éviter de les mélanger car certains peuvent ne pas être restitués correctement.

Les composants Swing utilisent des modèles pour contenir leurs états ou leurs données. Ces modèles sont des classes particulières qui possèdent toutes un comportement par défaut.

15.2. Les packages Swing

Swing contient plusieurs packages :

javax.swing	package principal : il contient les interfaces, les principaux composants, les modèles par défaut
javax.swing.border	Classes représentant les bordures
javax.swing.colorchooser	Classes définissant un composant pour la sélection de couleurs
javax.swing.event	Classes et interfaces pour les événements spécifiques à Swing. Les autres événements sont ceux d'AWT (java.awt.event)
javax.swing.filechooser	Classes définissant un composant pour la sélection de fichiers
javax.swing.plaf	Classes et interfaces génériques pour gérer l'apparence
javax.swing.plaf.basic	Classes et interfaces de base pour gérer l'apparence
javax.swing.plaf.metal	Classes et interfaces pour définir l'apparence Metal qui est l'apparence par défaut
javax.swing.table	Classes définissant un composant pour la présentation de données sous forme de tableau
javax.swing.text	Classes et interfaces de bases pour les composants manipulant du texte
javax.swing.text.html	Classes permettant le support du format HTML
javax.swing.text.html.parser	Classes permettant d'analyser des données au format HTML
javax.swing.text.rtf	Classes permettant le support du format RTF
javax.swing.tree	Classes définissant un composant pour la présentation de données sous forme d'arbre
javax.swing.undo	Classes permettant d'implémenter les fonctions annuler/refaire

15.3. Un exemple de fenêtre autonome

La classe de base d'une application est la classe JFrame. Son rôle est équivalent à la classe Frame de l'AWT et elle s'utilise de la même façon.

Exemple (code Java 1.1) :

```
import javax.swing.*;
import java.awt.event.*;

public class swing1 extends JFrame {

   public swing1() {
      super("titre de l'application");
```

```
      WindowListener l = new WindowAdapter() {
         public void windowClosing(WindowEvent e){
            System.exit(0);
         }
      };

      addWindowListener(l);
      setSize(200,100);
      setVisible(true);
   }

   public static void main(String [] args){
      JFrame frame = new swing1();
   }
}
```

15.4. Les composants Swing

Il existe des composants Swing équivalents pour chacun des composants AWT avec des constructeurs semblables. De nombreux constructeurs acceptent comme argument un objet de type Icon, qui représente une petite image généralement stockée au format Gif.

Le constructeur d'un objet Icon admet comme seul paramètre le nom ou l'URL d'un fichier graphique

Exemple (code Java 1.1) :

```
import javax.swing.*;
import java.awt.event.*;

public class swing3 extends JFrame {

  public swing3() {

     super("titre de l'application");

      WindowListener l = new WindowAdapter() {
         public void windowClosing(WindowEvent e){
            System.exit(0);
         }
      };

      addWindowListener(l);

      ImageIcon img = new ImageIcon("tips.gif");
      JButton bouton = new JButton("Mon bouton",img);

      JPanel panneau = new JPanel();
      panneau.add(bouton);
      setContentPane(panneau);
      setSize(200,100);
      setVisible(true);
   }

  public static void main(String [] args){
      JFrame frame = new swing3();
  }
}
```

15.4.1. La classe JFrame

JFrame est l'équivalent de la classe Frame de l'AWT : les principales différences sont l'utilisation du double buffering qui améliore les rafraichissements et l'utilisation d'un panneau de contenu (contentPane) pour insérer des composants (ils ne sont plus insérés directement au JFrame mais à l'objet contentPane qui lui est associé). Elle représente une fenêtre principale qui possède un titre, une taille modifiable et éventuellement un menu.

La classe possède plusieurs constructeurs :

Constructeur	Rôle
JFrame()	
JFrame(String)	Création d'une instance en précisant le titre

Par défaut, la fenêtre créée n'est pas visible. La méthode setVisible() permet de l'afficher.

Exemple (code Java 1.1) :

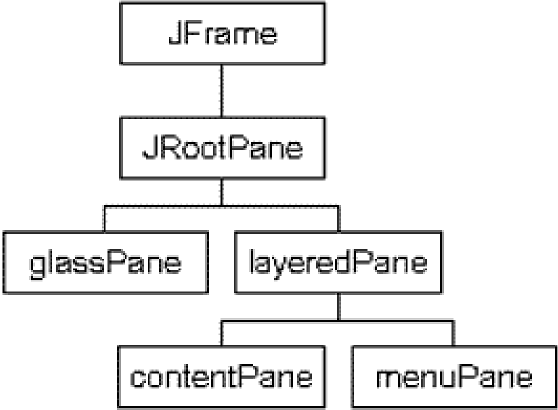
```
import javax.swing.*;

public class TestJFrame1 {

  public static void main(String argv[]) {
    JFrame f = new JFrame("ma fenetre");
    f.setSize(300,100);
    f.setVisible(true);
  }
}
```

La gestion des événements est identique à celle utilisée dans l'AWT depuis le J.D.K. 1.1.

Exemple (code Java 1.1) :

```
import javax.swing.*;
import java.awt.event.*;

public class swing2 extends JFrame {

   public swing2() {

     super("titre de l'application");

      WindowListener l = new WindowAdapter() {
         public void windowClosing(WindowEvent e){
           System.exit(0);
         }
      };
     addWindowListener(l);

     JButton bouton = new JButton("Mon bouton");
     JPanel panneau = new JPanel();
     panneau.add(bouton);

     setContentPane(panneau);
     setSize(200,100);
     setVisible(true);
   }

   public static void main(String [] args){
      JFrame frame = new swing2();
   }
}
```

Tous les composants associés à un objet JFrame sont gérés par un objet de la classe JRootPane. Un objet JRootPane contient plusieurs Panes. Tous les composants ajoutés au JFame doivent être ajoutés à un des Pane du JRootPane et non au JFrame directement. C'est aussi à un de ces Panes qu'il faut associer un layout manager si nécessaire.



Le Pane le plus utilisé est le ContentPane. Le Layout manager par défaut du contentPane est BorderLayout. Il est possible de le changer :

Exemple (code Java 1.1) :

```
...
f.getContentPane().setLayout(new FlowLayout());
...
```

Exemple (code Java 1.1) :

```
import javax.swing.*;

public class TestJFrame2 {

  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");
    f.setSize(300,100);
    JButton b =new JButton("Mon bouton");
    f.getContentPane().add(b);
    f.setVisible(true);
  }
}
```

Le JRootPane se compose de plusieurs éléments :

- glassPane : un JPanel par défaut
- layeredPane qui se compose du contentPane (un JPanel par défaut) et du menuBar (un objet de type JMenuBar)

Le glassPane est un JPanel transparent qui se situe au dessus du layeredPane. Le glassPane peut être n'importe quel composant : pour le modifier il faut utiliser la méthode setGlassPane() en fournissant le composant en paramètre.

Le layeredPane regroupe le contentPane et le menuBar.

Le contentPane est par défaut un JPanel opaque dont le gestionnaire de présentation est un BorderLayout. Ce panel peut être remplacé par n'importe quel composant grâce à la méthode setContentPane().

Attention : il ne faut pas utiliser directement la méthode setLayout() d'un objet JFrame sinon une exception est levée.

Exemple (code Java 1.1) :

```
import javax.swing.*;
import java.awt.*;
```

```
public class TestJFrame7 {

  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");
    f.setLayout(new FlowLayout());
    f.setSize(300,100);
    f.setVisible(true);
  }
}
```

Résultat :

```
C:\swing\code>java TestJFrame7
Exception in thread "main" java.lang.Error: Do not use javax.swing.JFrame.setLay
out() use javax.swing.JFrame.getContentPane().setLayout() instead
        at javax.swing.JFrame.createRootPaneException(Unknown Source)
        at javax.swing.JFrame.setLayout(Unknown Source)
        at TestJFrame7.main(TestJFrame7.java:8)
```

Le menuBar permet d'attacher un menu à la JFrame. Par défaut, le menuBar est vide. La méthode setJMenuBar() permet d'affecter un menu à la JFrame.

Exemple (code Java 1.1) : Création d'un menu très simple

```
import javax.swing.*;
import java.awt.*;

public class TestJFrame6 {

  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");
    f.setSize(300,100);
    JButton b =new JButton("Mon bouton");
    f.getContentPane().add(b);

    JMenuBar menuBar = new JMenuBar();
    f.setJMenuBar(menuBar);

    JMenu menu = new JMenu("Fichier");
    menu.add(menuItem);
    menuBar.add(menu);

    f.setVisible(true);
  }
}
```

15.4.1.1. Le comportement par défaut à la fermeture

Il est possible de préciser comment un objet JFrame, JInternalFrame, ou JDialog réagit à sa fermeture grâce à la méthode setDefaultCloseOperation(). Cette méthode attend en paramètre une valeur qui peut être :

Constante	Rôle
WindowConstants.DISPOSE_ON_CLOSE	détruit la fenêtre
WindowConstants.DO_NOTHING_ON_CLOSE	rend le bouton de fermeture inactif
WindowConstants.HIDE_ON_CLOSE	cache la fenêtre

Cette méthode ne permet pas d'associer d'autres traitements. Dans ce cas, il faut intercepter l'événement et lui associer les traitements.

Exemple (code Java 1.1) : la fenêtre disparaît lors de sa fermeture mais l'application ne se termine pas.

```
import javax.swing.*;

public class TestJFrame3 {

  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");
    f.setSize(300,100);
    JButton b =new JButton("Mon bouton");
    f.getContentPane().add(b);

    f.setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);

    f.setVisible(true);
  }
}
```

15.4.1.2. La personnalisation de l'icône

La méthode setIconImage() permet de modifier l'icône de la JFrame.

Exemple (code Java 1.1) :

```
import javax.swing.*;

public class TestJFrame4 {

  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");
    f.setSize(300,100);
    JButton b =new JButton("Mon bouton");
    f.getContentPane().add(b);
    f.setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);

    ImageIcon image = new ImageIcon("book.gif");
    f.setIconImage(image.getImage());
    f.setVisible(true);

  }
}
```

Si l'image n'est pas trouvée, alors l'icône est vide. Si l'image est trop grande, elle est redimensionnée.

15.4.1.3. Centrer une JFrame à l'écran

Par défaut, une JFrame est affichée dans le coin supérieur gauche de l'écran. Pour la centrer dans l'écran, il faut procéder comme pour une Frame : déterminer la position de la Frame en fonction de sa dimension et de celle de l'écran et utiliser la méthode setLocation() pour affecter cette position.

Exemple (code Java 1.1) :

```
import javax.swing.*;
import java.awt.*;

public class TestJFrame5 {

  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");
    f.setSize(300,100);
    JButton b =new JButton("Mon bouton");
    f.getContentPane().add(b);

    f.setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);

    Dimension dim = Toolkit.getDefaultToolkit().getScreenSize();
    f.setLocation(dim.width/2 - f.getWidth()/2, dim.height/2 - f.getHeight()/2);

    f.setVisible(true);
  }
}
```

15.4.1.4. Les événements associées à un JFrame

La gestion des événements associés à un objet JFrame est identique à celle utilisée pour un objet de type Frame de AWT.

Exemple (code Java 1.1) :

```
import javax.swing.*;
import java.awt.event.*;

public class TestJFrame8 {

  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");
    f.setSize(300,100);
    f.setVisible(true);
    f.addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
          System.exit(0);
        }
      }
    );

  }
}
```

15.4.2. Les étiquettes : la classe JLabel

Le composant JLabel propose les mêmes fonctionnalités que les intitulés AWT mais ils peuvent en plus contenir des icônes .

Cette classe possède plusieurs constructeurs :

Constructeurs	Rôle
JLabel()	Création d'une instance sans texte ni image
JLabel(Icon)	Création d'une instance en précisant l'image

JLabel(Icon, int)	Création d'une instance en précisant l'image et l'alignement horizontal
JLabel(String)	Création d'une instance en précisant le texte
JLabel(String, Icon, int)	Création d'une instance en précisant le texte, l'image et l'alignement horizontal
JLabel(String, int)	Création d'une instance en précisant le texte et l'alignement horizontal

Le composant JLabel permet d'afficher un texte et/ou une icône en précisant leur alignement. L'icône doit être au format GIF et peut être une animation dans ce format.

Exemple (code Java 1.1) :

```
import javax.swing.*;
import java.awt.*;

public class TestJLabel1 {

  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");
    f.setSize(100,200);

    JPanel pannel = new JPanel();
    JLabel jLabel1 =new JLabel("Mon texte dans JLabel");
    pannel.add(jLabel1);

    ImageIcon icone = new ImageIcon("book.gif");
    JLabel jLabel2 =new JLabel(icone);
    pannel.add(jLabel2);

    JLabel jLabel3 =new JLabel("Mon texte",icone,SwingConstants.LEFT);
    pannel.add(jLabel3);

    f.getContentPane().add(pannel);
    f.setVisible(true);
  }
}
```

La classe JLabel définit plusieurs méthodes pour modifier l'apparence du composant :

Méthodes	Rôle
setText()	Permet d'initialiser ou de modifier le texte affiché
setOpaque()	Indique si le composant est transparent (paramètre false) ou opaque (true)
setBackground()	Indique la couleur de fond du composant (setOpaque doit être à true)
setFont()	Permet de préciser la police du texte
setForeGround()	Permet de préciser la couleur du texte
setHorizontalAlignment()	Permet de modifier l'alignement horizontal du texte et de l'icône
setVerticalAlignment()	Permet de modifier l'alignement vertical du texte et de l'icône
setHorizontalTextAlignment()	Permet de modifier l'alignement horizontal du texte uniquement
setVerticalTextAlignment()	Permet de modifier l'alignement vertical du texte uniquement Exemple : jLabel.setVerticalTextPosition(SwingConstants.TOP);
setIcon()	Permet d'assigner une icône
setDisabledIcon()	Permet d'assigner une icône dans un état désactivée

L'alignement vertical par défaut d'un JLabel est centré. L'alignement horizontal par défaut est soit à droite si il ne contient que du texte, soit centré si il contient une image avec ou sans texte. Pour modifier cet alignement, il suffit d'utiliser les méthodes ci dessus en utilisant des constantes en paramètres : SwingConstants.LEFT, SwingConstants.CENTER, SwingConstants.RIGHT, SwingConstants.TOP, SwingConstants.BOTTOM

Par défaut, un JLabel est transparent : son fond n'est pas dessiné. Pour le dessiner, il faut utiliser la méthode setOpaque() :

Exemple (code Java 1.1) :

```
import javax.swing.*;
import java.awt.*;

public class TestJLabel2 {

  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");

    f.setSize(100,200);
    JPanel pannel = new JPanel();

    JLabel jLabel1 =new JLabel("Mon texte dans JLabel 1");
    jLabel1.setBackground(Color.red);
    pannel.add(jLabel1);

    JLabel jLabel2 =new JLabel("Mon texte dans JLabel 2");
    jLabel2.setBackground(Color.red);
    jLabel2.setOpaque(true);
    pannel.add(jLabel2);

    f.getContentPane().add(pannel);
    f.setVisible(true);
  }
}
```

Dans l'exemple, les 2 JLabel ont le fond rouge demandé par la méthode setBackground(). Seul le deuxième affiche un fond rouge car il est rendu opaque avec la méthode setOpaque().

Il est possible d'associer un raccourci clavier au JLabel qui permet de donner le focus à un autre composant. La méthode setDisplayedMnemonic() permet de définir le raccourci clavier. Celui ci sera activé en utilisant la touche Alt avec le caractère fourni en paramètre. La méthode setLabelFor() permet d'associer le composant fourni en paramètre au raccourci.

Exemple (code Java 1.1) :

```
import javax.swing.*;
import java.awt.*;

public class TestJLabel3 {

  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");
    f.setSize(300,100);
    JPanel pannel = new JPanel();

    JButton bouton = new JButton("saisir");
    pannel.add(bouton);

    JTextField jEdit = new JTextField("votre nom");

    JLabel jLabel1 =new JLabel("Nom : ");
    jLabel1.setBackground(Color.red);
    jLabel1.setDisplayedMnemonic('n');
    jLabel1.setLabelFor(jEdit);
```

```
    pannel.add(jLabel1);
    pannel.add(jEdit);

    f.getContentPane().add(pannel);
    f.setVisible(true);
  }
}
```

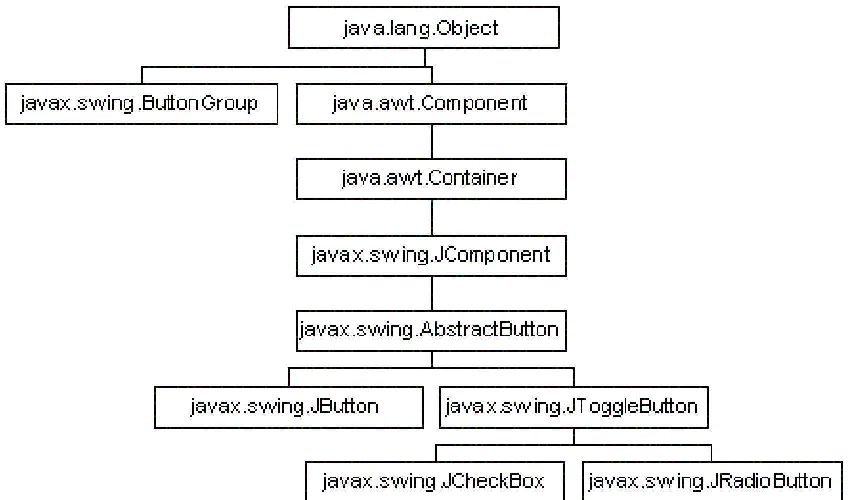
Dans l'exemple, à l'ouverture de la fenêtre, le focus est sur le bouton. Un appui sur Alt+'n' donne le focus au champ de saisie.

15.4.3. Les panneaux : la classe JPanel

La classe JPanel est un conteneur utilisé pour regrouper et organiser des composants grâce à un gestionnaire de présentation (layout manager). Le gestionnaire par défaut d'un JPanel est un objet de la classe FlowLayout.

15.5. Les boutons

Il existe plusieurs boutons définis par Swing.



15.5.1. La classe AbstractButton

C'est une classe abstraite dont héritent les boutons Swing JButton, JMenuItem et JToggleButton.

Cette classe définit de nombreuses méthodes dont les principales sont :

Méthode	Rôle
AddActionListener	Associer un écouteur sur un événement de type ActionEvent
AddChangeListener	Associer un écouteur sur un événement de type ChangeEvent
AddItemListener	Associer un écouteur sur un événement de type ItemEvent
doClick()	Déclencher un clic par programmation
getText()	Obtenir le texte affiché par le composant
setDisabledIcon()	Associer une icône affichée lorsque le composant à l'état désélectionné

setDisabledSelectedIcon()	Associer une icône affichée lors du passage de la souris sur le composant à l'état désélectionné
setEnabled()	Activer/désactiver le composant
setMnemonic()	Associer un raccourci clavier
setPressedIcon()	Associer une icône affichée lorsque le composant est cliqué
setRolloverIcon()	Associer une icône affichée lors du passage de la souris sur le composant
setRolloverSelectedIcon()	Associer une icône affichée lors du passage de la souris sur le composant à l'état sélectionné
setSelectedIcon()	Associer une icône affichée lorsque le composant à l'état sélectionné
setText()	Mettre à jour le texte du composant
isSelected()	Indiquer si le composant est dans l'état sélectionné
setSelected()	Mettre à jour l'état sélectionné du composant

Tous les boutons peuvent afficher du texte et/ou une image.

Il est possible de préciser une image différente lors du passage de la souris sur le composant et lors de l'enfoncement du bouton : dans ce cas, il faut créer trois images pour chacun des états (normal, enfoncé et survolé). L'image normale est associée au bouton grâce au constructeur, l'image enfoncée grâce à la méthode setPressedIcon() et l'image lors d'un survole grâce à la méthode setRolloverIcon(). Il suffit enfin d'appeler la méthode setRolloverEnable() avec en paramètre la valeur true.

Exemple (code Java 1.1) :

```
import javax.swing.*;
import java.awt.event.*;

public class swing4 extends JFrame {

   public swing4() {
     super("titre de l'application");

     WindowListener l = new WindowAdapter() {
       public void windowClosing(WindowEvent e){
         System.exit(0);
       }
     };
     addWindowListener(l);

     ImageIcon imageNormale = new ImageIcon("arrow.gif");
     ImageIcon imagePassage =  new ImageIcon("arrowr.gif");
     ImageIcon imageEnfoncee = new ImageIcon("arrowy.gif");

     JButton bouton = new JButton("Mon bouton",imageNormale);
     bouton.setPressedIcon(imageEnfoncee);
     bouton.setRolloverIcon(imagePassage);
     bouton.setRolloverEnabled(true);
     getContentPane().add(bouton, "Center");

     JPanel panneau = new JPanel();
     panneau.add(bouton);
     setContentPane(panneau);
     setSize(200,100);
     setVisible(true);
   }

   public static void main(String [] args){
      JFrame frame = new swing4();
   }
}
```

Un bouton peut recevoir des événements de type ActionEvents (le bouton a été activé), ChangeEvents, et ItemEvents.

Exemple (code Java 1.1) : fermeture de l'application lors de l'activation du bouton

```
import javax.swing.*;
import java.awt.event.*;

public class TestJButton3 {

  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");
    f.setSize(300,100);
    JPanel pannel = new JPanel();

    JButton bouton1 = new JButton("Bouton1");
    bouton1.addActionListener( new ActionListener() {
       public void actionPerformed(ActionEvent e) {
         System.exit(0);
       }
    }
    );

    pannel.add(bouton1);
    f.getContentPane().add(pannel);
    f.setVisible(true);
  }
}
```

Pour de plus amples informations sur la gestion des événements, voir le chapitre correspondant.

15.5.2. La classe JButton

JButton est un composant qui représente un bouton : il peut contenir un texte et/ou une icône.

Les constructeurs sont :

Constructeur	Rôle
JButton()	
JButton(String)	préciser le texte du bouton
JButton(Icon)	préciser une icône
JButton(String, Icon)	préciser un texte et une icone

Il ne gère pas d'état. Toutes les indications concernant le contenu du composant JLabel sont valables pour le composant JButton.

Exemple (code Java 1.1) : un bouton avec une image

```
import javax.swing.*;
import java.awt.event.*;

public class swing3 extends JFrame {

   public swing3() {

      super("titre de l'application");

      WindowListener l = new WindowAdapter() {
         public void windowClosing(WindowEvent e){
            System.exit(0);
         }
```

```
    };
    addWindowListener(l);

    ImageIcon img = new ImageIcon("tips.gif");
    JButton bouton = new JButton("Mon bouton",img);

    JPanel panneau = new JPanel();
    panneau.add(bouton);
    setContentPane(panneau);
    setSize(200,100);
    setVisible(true);
  }

  public static void main(String [] args){
    JFrame frame = new swing3();
  }

}
```

L'image gif peut être une animation.

Dans un conteneur de type JRootPane, il est possible de définir un bouton par défaut grâce à sa méthode setDefaultButton().

Exemple (code Java 1.1) : définition d'un bouton par défaut dans un JFrame

```
import javax.swing.*;
import java.awt.*;

public class TestJButton2 {

  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");
    f.setSize(300,100);
    JPanel pannel = new JPanel();
    JButton bouton1 = new JButton("Bouton 1");
    pannel.add(bouton1);

    JButton bouton2 = new JButton("Bouton 2");
    pannel.add(bouton2);

    JButton bouton3 = new JButton("Bouton 3");
    pannel.add(bouton3);

    f.getContentPane().add(pannel);
    f.getRootPane().setDefaultButton(bouton3);
    f.setVisible(true);
  }
}
```

Le bouton par défaut est activé par un appui sur la touche Entrée alors que le bouton actif est activé par un appui sur la barre d'espace.

La méthode isDefaultButton() de JButton permet de savoir si le composant est le bouton par défaut.

15.5.3. La classe JToggleButton

Cette classe définit un bouton à deux états : c'est la classe mère des composants JCheckBox et JRadioButton.

La méthode setSelected() héritée de AbstractButton permet de mettre à jour l'état du bouton. La méthode isSelected() permet de connaître cet état.

15.5.4. La classe ButtonGroup

La classe ButtonGroup permet de gérer un ensemble de boutons en garantissant qu'un seul bouton du groupe sera sélectionné.

Pour utiliser la classe ButtonGroup, il suffit d'instancier un objet et d'ajouter des boutons (objets héritant de la classe AbstractButton) grâce à la méthode add(). Il est préférable d'utiliser des objets de la classe JToggleButton ou d'une de ces classes filles car elles sont capables de gérer leurs états.

Exemple (code Java 1.1) :

```
import javax.swing.*;

public class TestGroupButton1 {

  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");
    f.setSize(300,100);
    JPanel pannel = new JPanel();

    ButtonGroup groupe = new ButtonGroup();
    JRadioButton bouton1 = new JRadioButton("Bouton 1");
    groupe.add(bouton1);
    pannel.add(bouton1);
    JRadioButton bouton2 = new JRadioButton("Bouton 2");
    groupe.add(bouton2);
    pannel.add(bouton2);
    JRadioButton bouton3 = new JRadioButton("Bouton 3");
    groupe.add(bouton3);
    pannel.add(bouton3);

    f.getContentPane().add(pannel);
    f.setVisible(true);
  }
}
```

15.5.5. Les cases à cocher : la classe JCheckBox

Les constructeurs sont les suivants :

Constructeur	Rôle
JCheckBox(String)	précise l'intitulé
JCheckBox(String, boolean)	précise l'intitulé et l'état
JCheckBox(Icon)	précise une icône comme intitulé
JCheckBox(Icon, boolean)	précise une icône comme intitulé et l'état
JCheckBox(String, Icon)	précise un texte et une icône comme intitulé
JCheckBox(String, Icon, boolean)	précise un texte et une icône comme intitulé et l'état

Un groupe de cases à cocher peut être défini avec la classe ButtonGroup. Dans ce cas, un seul composant du groupe peut être sélectionné. Pour l'utiliser, il faut créer un objet de la classe ButtonGroup et utiliser la méthode add() pour ajouter un composant au groupe.

Exemple (code Java 1.1) :

```
import javax.swing.*;

public class TestJCheckBox1 {
```

```
  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");
    f.setSize(300,100);
    JPanel pannel = new JPanel();

    JCheckBox bouton1 = new JCheckBox("Bouton 1");
    pannel.add(bouton1);
    JCheckBox bouton2 = new JCheckBox("Bouton 2");
    pannel.add(bouton2);
    JCheckBox bouton3 = new JCheckBox("Bouton 3");
    pannel.add(bouton3);

    f.getContentPane().add(pannel);
    f.setVisible(true);
  }
}
```

15.5.6. Les boutons radio : la classe JRadioButton

Les constructeurs sont les mêmes que ceux de la classe JCheckBox.

Exemple (code Java 1.1) :

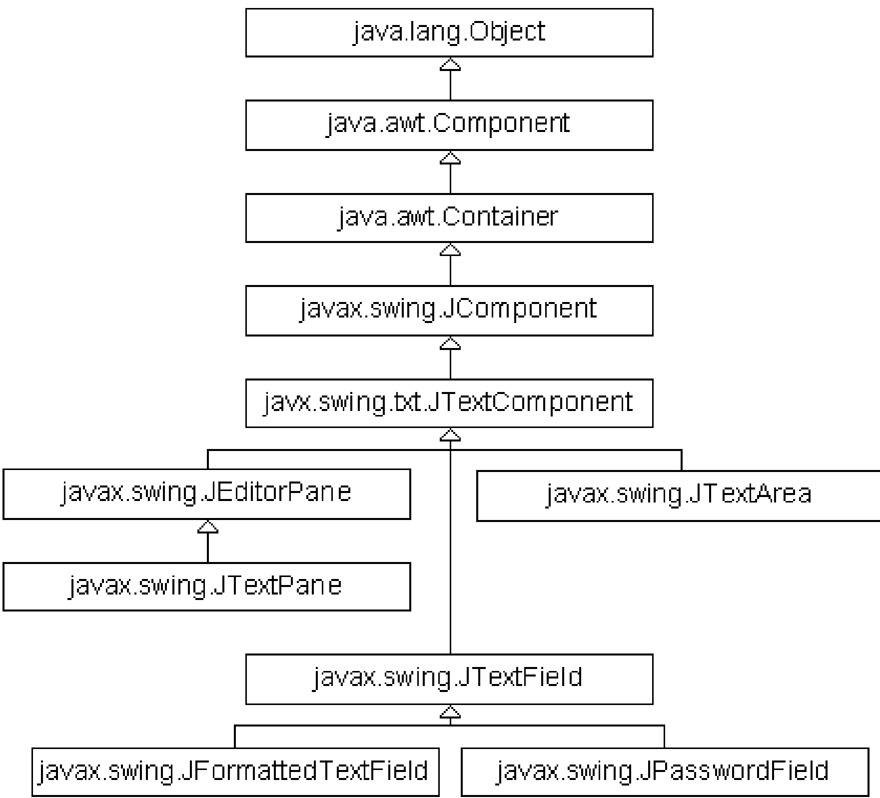
```
import javax.swing.*;

public class TestJRadioButton1 {

  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");
    f.setSize(300,100);
    JPanel pannel = new JPanel();
    JRadioButton bouton1 = new JRadioButton("Bouton 1");
    pannel.add(bouton1);
    JRadioButton bouton2 = new JRadioButton("Bouton 2");
    pannel.add(bouton2);
    JRadioButton bouton3 = new JRadioButton("Bouton 3");
    pannel.add(bouton3);

    f.getContentPane().add(pannel);
    f.setVisible(true);
  }
}
```

Pour regrouper plusieurs boutons radio, il faut utiliser la classe CheckboxGroup

La suite de ce section sera développée dans une version future de ce document

15.6. Les composants de saisie de texte

Swing possède plusieurs composants pour permettre la saisie de texte.



15.6.1. La classe JTextComponent

La classe abstraite JTextComponent est la classe mère de tout les composants permettant la saisie de texte.

Les données du composant (le modèle dans le motif de conception MVC) sont encapsulées dans un objet qui implémente l'interface Document. Deux classes implémentant cette interface sont fournies en standard : PlainDocument pour du texte simple et StyledDocument pour du texte riche pouvant contenir entre autre plusieurs polices de caractères, des couleurs, des images, ...

La classe JTextComponent possède de nombreuses méthodes dont les principales sont :

Méthode	Rôle
void copy()	Copier le contenu du texte et le mettre dans le presse papier système
void cut()	Couper le contenu du texte et le mettre dans le presse papier système
Document getDocument()	Renvoyer l'objet de type Document qui encapsule le texte saisi
String getSelectectedText()	Renvoyer le texte sélectionné dans le composant
int getSelectionEnd()	Renvoyer la position de la fin de la sélection
int getSelectionStart()	Renvoyer la position du début de la sélection
String getText()	Renvoyer le texte saisie
String getText(int, int)	Renvoyer une portion du texte incluse à partir de la position donnée par le premier paramètre et la longueur donnée dans le second paramètre
bool isEditable()	Renvoyer un booléen qui précise si le texte est éditable ou non
void paste()	Coller le contenu du presse papier système dans le composant
void select(int,int)	Sélectionner une portion du texte dont les positions de début et de fin sont fournies en

	paramètres
void setCaretPosition(int)	Déplacer le curseur à la position dans le texte précisé en paramètre
void setEditable(boolean)	Permet de préciser si les données du composant sont éditables ou non
void setSelectionEnd(int)	Modifier la position de la fin de la sélection
void setSelectionStart(int)	Modifier la position du début de la sélection
void setText(String)	Modifier le contenu du texte

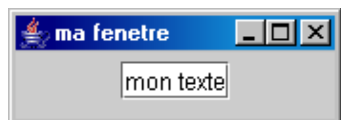
Toutes ces méthodes sont donc accessibles grâce à l'héritage pour tous les composants de saisie de texte proposés par Swing.

15.6.2. La classe JTextField

La classe javax.Swing.JTextField est un composant qui permet la saisie d'une seule ligne de texte simple. Son modèle utilise un objet de type PlainDocument.

Exemple (code Java 1.1) :

```
import javax.swing.*;

public class JTextField1 {

  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");
    f.setSize(300, 100);
    JPanel pannel = new JPanel();

    JTextField  testField1 = new JTextField ("mon texte");

    pannel.add(testField1);
    f.getContentPane().add(pannel);
    f.setVisible(true);
  }
}
```



La propriété horizontalAligment permet de préciser l'alignement du texte dans le composant en utilisant les valeurs JTextField.LEFT , JTextField.CENTER ou JTextField.RIGHT.

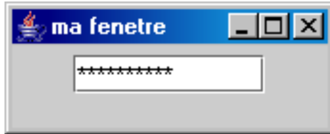
15.6.3. La classe JPasswordField

La classe JPasswordField permet la saisie d'un texte dont tous les caractères saisis seront affichés sous la forme d'un caractère particulier ('*' par défaut). Cette classe hérite de la classe JTextField.

Exemple (code Java 1.1) :

```
import java.awt.Dimension;

import javax.swing.*;

public class JPasswordField1 {
```

```
  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");
    f.setSize(300, 100);
    JPanel pannel = new JPanel();

    JPasswordField  passwordField1 = new JPasswordField ("");
    passwordField1.setPreferredSize(new Dimension(100,20 ));

    pannel.add(passwordField1);
    f.getContentPane().add(pannel);
    f.setVisible(true);
  }
}
```



La méthode setEchoChar(char) permet de préciser le caractère qui sera utilisé pour afficher la saisie d'un caractère.

Il ne faut pas utiliser la méthode getText() qui est déclarée deprecated mais la méthode getPassword() pour obtenir la valeur du texte saisi.

Exemple (code Java 1.1) :

```
import java.awt.Dimension;
import java.awt.event.*;

import javax.swing.*;

public class JPasswordField2 implements ActionListener {

  JPasswordField passwordField1 = null;

  public static void main(String argv[]) {
    JPasswordField2 jpf2 = new JPasswordField2();
    jpf2.init();
  }

  public void init() {
    JFrame f = new JFrame("ma fenetre");
    f.setSize(300, 100);
    JPanel pannel = new JPanel();

    passwordField1 = new JPasswordField("");
    passwordField1.setPreferredSize(new Dimension(100, 20));
    pannel.add(passwordField1);

    JButton bouton1 = new JButton("Afficher");
    bouton1.addActionListener(this);

    pannel.add(bouton1);
    f.getContentPane().add(pannel);
    f.setVisible(true);
  }

  public void actionPerformed(ActionEvent e) {

    System.out.println("texte saisie = " + String.copyValueOf(passwordField1.getPassword()));
  }

}
```

Les méthodes copy() et cut() sont redéfinies pour empêcher l'envoi du contenu dans le composant et émettre simplement un beep.

15.6.4. La classe JFormattedTextField

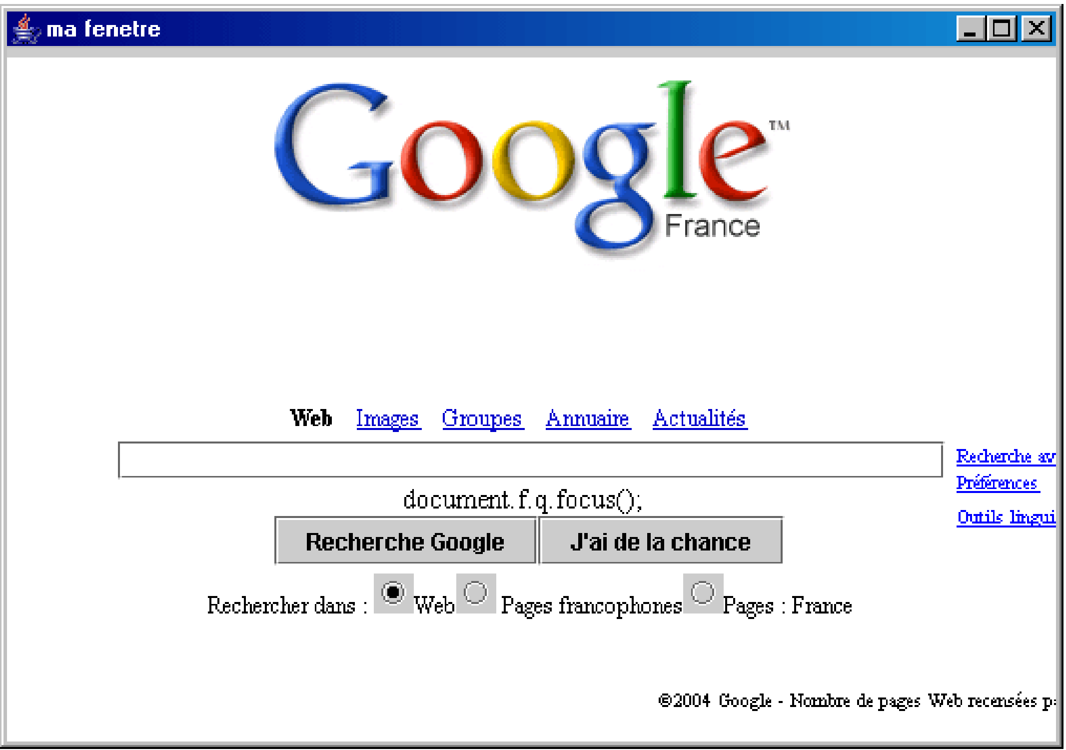
Le JDK 1.4 propose la classe JFormattedTextField pour faciliter la création d'un composant de saisie personnalisé. Cette classe hérite de la classe JTextField.

15.6.5. La classe JEditorPane

Ce composant permet de saisie de texte riche multi-lignes. Ce type de texte peut contenir des informations de mise en pages et de formatage. En standard, Swing propose le support des formats RTF et HTML.

Exemple (code Java 1.1) : affichage de la page de Google avec gestion des hyperliens

```
import java.net.URL;
import javax.swing.*;
import javax.swing.event.*;

public class JEditorPane1 {

  public static void main(String[] args) {
    final JEditorPane editeur;
    JPanel pannel = new JPanel();

    try {
      editeur = new JEditorPane(new URL("http://google.fr"));
      editeur.setEditable(false);
      editeur.addHyperlinkListener(new HyperlinkListener() {
        public void hyperlinkUpdate(HyperlinkEvent e) {
          if (e.getEventType() == HyperlinkEvent.EventType.ACTIVATED) {
            URL url = e.getURL();
            if (url == null)
              return;
            try {
              editeur.setPage(e.getURL());
            } catch (Exception ex) {
              ex.printStackTrace();
            }
          }
        }
      });

      pannel.add(editeur);
    } catch (Exception e1) {
      e1.printStackTrace();
    }
    JFrame f = new JFrame("ma fenetre");
    f.setSize(500, 300);

    f.getContentPane().add(pannel);
    f.setVisible(true);

  }
}
```



15.6.6. La classe JTextPane

La suite de cette section sera développée dans une version future de ce document

15.6.7. La classe JTextArea

La classe JTextArea est un composant qui permet la saisie de texte simple en mode multi-lignes. Le modèle utilisé par ce composant est le PlainDocument : il ne peut donc contenir que du texte brut sans éléments multiples de formatage.

JTexteArea propose plusieurs méthodes pour ajouter du texte dans son modèle :

- soit fournir le texte en paramètre du constructeur utilisé
- soit utiliser la méthode setText() qui permet d'initialiser le texte du composant
- soit utiliser la méthode append() qui permet d'ajouter du texte à la fin de celui contenu dans le texte du composant
- soit utiliser la méthode insert() permet d'insérer un texte à une position données en caractères dans le texte du composant

La méthode replaceRange() permet de remplacer une partie du texte désignée par la position du caractère de début et la position de son caractère de fin par le texte fourni en paramètre.

La propriété rows permet de définir le nombre de ligne affichée par le composant : cette propriété peut donc être modifiée lors d'un redimensionnement du composant. La propriété lineCount en lecture seule permet de savoir le nombre de lignes dont le texte est composé. Il ne faut pas confondre ces deux propriétés.

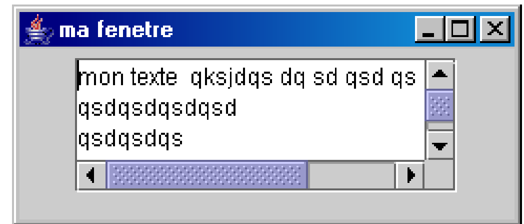
Exemple (code Java 1.1) :

```
import javax.swing.*;

public class JTextArea1 {

  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");
    f.setSize(300, 100);
    JPanel pannel = new JPanel();

    JTextArea  textArea1 = new JTextArea ("mon texte");

    pannel.add(textArea1);
    f.getContentPane().add(pannel);
    f.setVisible(true);
  }
}
```

Par défaut, la taille du composant augmente au fur et à mesure de l'augmentation de la taille du texte qu'il contient. Pour éviter cet effet, il faut encapsuler le JTextArea dans un JScrollPane.

Exemple (code Java 1.1) :

```
import java.awt.Dimension;
import javax.swing.*;

public class JTextArea1 {

  public static void main(String argv[]) {

    JFrame f = new JFrame("ma fenetre");
    f.setSize(300, 100);
    JPanel pannel = new JPanel();
    JTextArea  textArea1 = new JTextArea ("mon texte");
    JScrollPane scrollPane = new JScrollPane(textArea1);
    scrollPane.setPreferredSize(new Dimension(200,70));
    pannel.add(scrollPane);
    f.getContentPane().add(pannel);
    f.setVisible(true);
  }
}
```



15.7. Les onglets

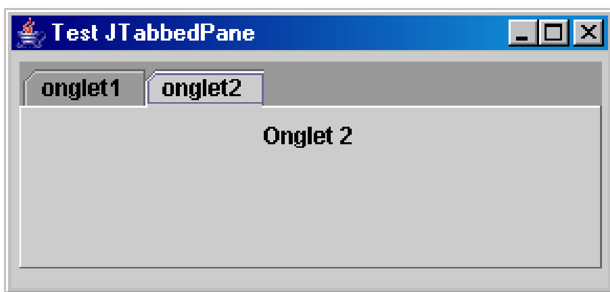
La classe javax.swing.JTabbedPane encapsule un ensemble d'onglets. Chaque onglet est constitué d'un titre, d'un composant et éventuellement d'une image.

Pour utiliser ce composant, il faut :

- instancier un objet de type JTabbedPane
- créer le composant de chaque onglet
- ajouter chaque onglet à l'objet JTabbedPane en utilisant la méthode addTab()

Exemple (code Java 1.1) :

```
import java.awt.Dimension;
import java.awt.event.KeyEvent;

import javax.swing.*;

public class TestJTabbedPane1 {

  public static void main(String[] args) {
    JFrame f = new JFrame("Test JTabbedPane");
    f.setSize(320, 150);
    JPanel pannel = new JPanel();

    JTabbedPane onglets = new JTabbedPane(SwingConstants.TOP);

    JPanel onglet1 = new JPanel();
    JLabel titreOnglet1 = new JLabel("Onglet 1");
    onglet1.add(titreOnglet1);
    onglet1.setPreferredSize(new Dimension(300, 80));
    onglets.addTab("onglet1", onglet1);

    JPanel onglet2 = new JPanel();
    JLabel titreOnglet2 = new JLabel("Onglet 2");
    onglet2.add(titreOnglet2);
    onglets.addTab("onglet2", onglet2);

    onglets.setOpaque(true);
    pannel.add(onglets);
    f.getContentPane().add(pannel);
    f.setVisible(true);

  }
}
```



A partir du JDK 1.4, il est possible d'ajouter un raccourci clavier sur chacun des onglets en utilisant la méthode setMnemonicAt(). Cette méthode attend deux paramètres : l'index de l'onglet concerné (le premier commence à 0) et la touche du clavier associée sous la forme d'une constance KeyEvent.VK_xxx. Pour utiliser ce raccourci, il suffit d'utiliser la touche désignée en paramètre de la méthode avec la touche Alt.

La classe JTabbedPane possède plusieurs méthodes qui permettent de définir le contenu de l'onglet :

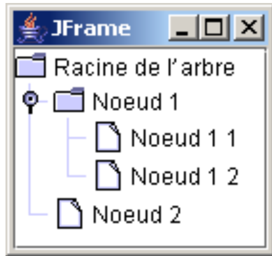
Méthodes	Rôles

addTab(String, Component)	Permet d'ajouter un nouvel onglet dont le titre et le composant sont fournis en paramètres. Cette méthode possède plusieurs surcharges qui permettent de préciser une icône et une bulle d'aide
insertTab(String, Icon, Component, String, index)	Permet d'insérer un onglet dont la position est précisée dans le dernier paramètre
remove(int)	Permet de supprimer l'onglet dont l'index est fourni en paramètre
setTabPlacement	Permet de préciser le positionnement des onglets dans le composant JTabbedPane. Les valeurs possibles sont les constantes TOP, BOTTOM, LEFT et RIGHT définies dans la classe JTabbedPane.

La méthode getSelectedIndex() permet d'obtenir l'index de l'onglet courant. La méthode setSelectedIndex() permet de définir l'onglet courant.

15.8. Le composant JTree

Le composant JTree permet de présenter des données sous une forme hiérarchique arborescente.



Aux premiers abords, le composant JTree peut sembler compliqué à mettre en oeuvre mais la compréhension de son mode de fonctionnement peut grandement faciliter son utilisation.

Il utilise le modèle MVC en proposant une séparation des données (data models) et du rendu de ces données (cell renderers).

Dans l'arbre, les éléments qui ne possèdent pas d'élément fils sont des feuilles (leaf). Chaque élément est associé à un objet (user object) qui va permettre de déterminé le libellé affiché dans l'arbre en utilisant la méthode toString().

15.8.1. La création d'une instance de la classe JTree

La classe JTree possède 7 constructeurs dont tous ceux qui attendent au moins un paramètre acceptent une collection pour initialiser tout ou partie du modèle de données de l'arbre :

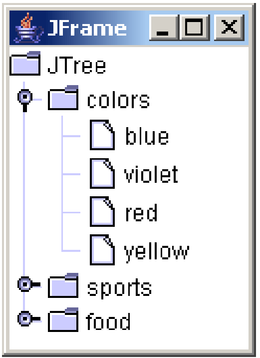
public JTree();
public JTree(Hashtable value);
public JTree(Vector value);
public JTree(Object[] value);
public JTree(TreeModel model);
public JTree(TreeNode rootNode);
public JTree(TreeNode rootNode, boolean askAllowsChildren);

Lorsqu'une instance de JTree est créée avec le constructeur par défaut, l'arbre obtenu contient des données par défaut.

Exemple (code Java 1.1) :

```
import javax.swing.JFrame;
import javax.swing.JTree;
```

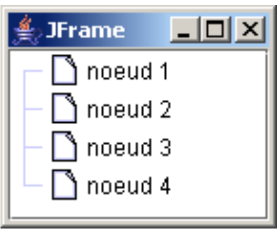
```
public class TestJtree extends JFrame {

  private javax.swing.JPanel jContentPane = null;
  private JTree              jTree        = null;

  private JTree getJTree() {
    if (jTree == null) {
      jTree = new JTree();
    }
    return jTree;
  }

  public static void main(String[] args) {
    TestJtree testJtree = new TestJtree();
    testJtree.setVisible(true);
  }

  public TestJtree() {
    super();
    initialize();
  }

  private void initialize() {
    this.setSize(300, 200);
    this.setContentPane(getJContentPane());
    this.setTitle("JFrame");
  }

  private javax.swing.JPanel getJContentPane() {
    if (jContentPane == null) {
      jContentPane = new javax.swing.JPanel();
      jContentPane.setLayout(new java.awt.BorderLayout());
      jContentPane.add(getJTree(), java.awt.BorderLayout.CENTER);
    }
    return jContentPane;
  }
}
```



Les trois constructeurs qui attendent en paramètre une collection permettent de créer un arbre avec une racine non affichée qui va contenir comme noeuds fils directs tous les éléments contenus dans la collection.

Exemple (code Java 1.1) :

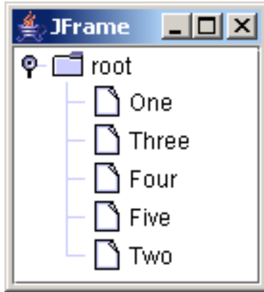
```
      String[] = {"noeud 1","noeud 2","noeud3","noeud 4"};
      jTree = new JTree(racine);
```



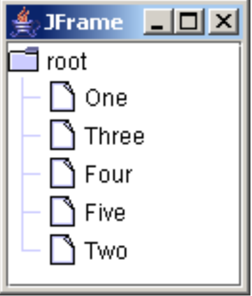
Dans ce cas, la racine n'est pas affichée. Pour l'afficher, il faut utiliser la méthode setRootVisible()

Exemple (code Java 1.1) :

```
jTree.setRootVisible(true);
```



Dans ce cas elle se nomme root et possède un commutateur qui permet de refermer ou d'étendre la racine. Pour supprimer ce commutateur, il faut utiliser la méthode jTree.setShowsRootHandles(false)



L'utilisation de l'une ou l'autre des collections n'est pas équivalente. Par exemple, l'utilisation d'une hashtable ne garantie pas l'ordre des noeuds puisque cette collection ne gère pas par définition un ordre précis.

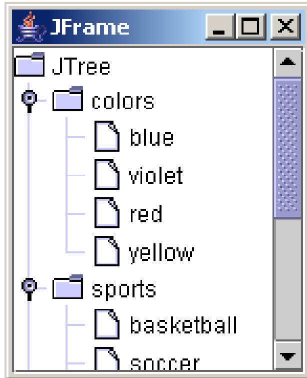
Généralement, la construction d'un arbre utilise un des constructeurs qui attend en paramètre un objet de type TreeModel ou TreeNode car ces deux objets permettent d'avoir un contrôle sur l'ensemble des données de l'arbre. Leur utilisation sera détaillée dans la section consacrée à la gestion des données de l'arbre

En fonction du nombre d'éléments et de l'état étendue ou non d'un ou plusieurs éléments, la taille de l'arbre peut varier : il est donc nécesssaire d'inclure le composant JTree dans un composant JScrollPane

Exemple (code Java 1.1) :

```
...

  private JScrollPane        jScrollPane  = null;

...

  private JScrollPane getJScrollPane() {
    if (jScrollPane == null) {
      jScrollPane = new JScrollPane();
      jScrollPane.setViewportView(getJTree());
    }
    return jScrollPane;
  }

...

  private javax.swing.JPanel getJContentPane() {
    if (jContentPane == null) {
      jContentPane = new javax.swing.JPanel();
      jContentPane.setLayout(new java.awt.BorderLayout());
      jContentPane.add(getJScrollPane(), java.awt.BorderLayout.CENTER);
    }
    return jContentPane;
```

```
  }
```



L'utilisateur peut sélectionner un noeud en cliquant sur son texte ou son icône. Un double clic sur le texte ou l'icône d'un noeud permet de l'étendre ou le refermer selon son état.

15.8.2. La gestion des données de l'arbre

Chaque arbre commence par un noeud racine. Par défaut, la racine et ces noeuds fils directs sont visibles. Chaque noeud de l'arbre peut avoir zéro ou plusieurs noeuds fils. Un noeud sans noeud fils est appelé un feuille de l'arbre (leaf)

En application du modèle MVC, le composant JTree ne gère pas directement chaque noeud et la façon dont ceux-ci sont organiser et stocker mais il utilise un objet dédié de type TreeModel.

Ainsi, comme dans d'autre composant Swing, le composant JTree manipule des objets implémentant des interfaces. Une classe qui encapsule les données de l'arbre doit implémenter l'interface TreeModel. Chaque noeud de l'arbre doit implémenter l'interface TreeNode.

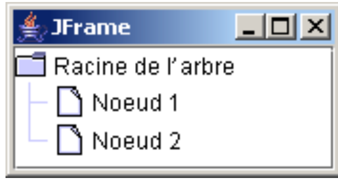
Pour préciser les données contenues dans l'arbre, il faut créer un objet qui va encapsuler ces données et les passer au constructeur de la classe Jtree. Cet objet peut être de type TreeNode ou TreeModel. Cet objet stocke les données de chaque noeud dans un objet de type TreeNode.

Généralement, le plus simple est de définir un type TreeNode personnalisé. Swing propose pour cela l'objet DefaultMutableTreeNode. Donc le plus simple est de créer une instance de type DefaultMutableTreeNode pour stocker les données et l'utiliser lors de l'appel du constructeur de la classe JTree.

La classe DefaultMutableTreeNode implémente l'interface MutableTreeNode qui elle-même hérite de l'interface TreeNode

Exemple (code Java 1.1) :

```
import javax.swing.tree.DefaultMutableTreeNode;

...

  private JTree getJTree() {

    if (jTree == null) {
      DefaultMutableTreeNode racine = new DefaultMutableTreeNode("Racine de l'arbre");
      DefaultMutableTreeNode noeud1 = new DefaultMutableTreeNode("Noeud 1");
      racine.add(noeud1);
      DefaultMutableTreeNode noeud2 = new DefaultMutableTreeNode("Noeud 2");
      racine.add(noeud2);
      jTree = new JTree(racine);
    }
    return jTree;
  }
```



Dans ce cas, une instance de la classe DefaultTreeModel est créée avec la racine fournie en paramètre du constructeur de la classe JTree.

Une autre solution permet de créer une instance de la classe DefaultTreeModel et de la passer en paramètre du constructeur de la classe JTree.

La méthode setModel() de la classe JTree permet d'associer un modèle de données à l'arbre.

15.8.2.1. L'interface TreeNode

Chaque noeud de l'arbre stocké dans le modèle de données implémente l'interface TreeNode.

Cette interface définit 7 méthodes dont la plupart concernent les relations entre les noeuds :

Méthode	Rôle
Enumeration children()	renvoie une collection des noeuds fils
boolean getAllowsChildren()	Renvoie un booléen qui precise si le noeud peut avoir des noeuds fils
TreeNode getChildAt(int index)	Renvoie le noeud fils correspondant à l'index fourni en paramètre
int getChildCount()	renvoie le nombre de noeuds fils direct du noeud
int getIndex(TreeNode child)	renvoie l'index du noeud passé en paramètre
TreeNode getParent()	renvoie le noeud père
boolean isLeaf()	renvoie un booléen qui précise si le noeud est une feuille

Chaque noeud ne peut avoir qu'un seul père (hormis le noeud racine qui ne possède pas de père) et autant de noeuds fils que souhaité. La méthode getParent() permet de renvoyer le noeud père. Elle renvoie null lorsque cette méthode est appelée sur le noeud racine.

La méthode getChildCount() renvoie le nombre de noeuds fils direct du noeud.

Si la méthode getAllowsChildren() permet de préciser si le noeud peut avoir des noeuds enfants : elle renvoie false alors le noeud sera toujours une feuille et ne pourra donc jamais avoir de noeuds fils.

La méthode isLeaf() renvoie un booléen précisant si le noeud est une feuille ou non. Une feuille est un noeud qui ne possède pas de noeud fils.

Les noeuds fils sont ordonnés car l'ordre de représentation des données peut être important dans la représentation de données hiérarchiques. La méthode getChildAt() renvoie le noeud fils dont l'index est fourni en paramètre de la méthode. La méthode getIndex() renvoie l'index du noeud fils passé en paramètre.

15.8.2.2. L'interface MutableTreeNode

Les 7 méthodes définies par l'interface TreeNode ne permettent que de lire des valeurs. Pour mettre à jour un noeud, il est nécéssaire d'utiliser l'interface MutableTreeNode qui hérite de la méthode TreeNode. Elle définit en plus plusieurs méthodes permettant de mettre à jour le noeud.

void insert(MutableTreeNode child, int index);
void remove(int index);
void remove(MutableTreeNode node);
void removeFromParent();
void setParent(MutableTreeNode parent);
void setUserObject(Object userObject);

La méthode insert() permet d'ajouter le noeud fourni en paramètre comme noeud fils à la position précisée par le second paramètre.

Il existe deux surcharges de la méthode remove() qui permet de déconnecter un noeud fils de son père. La première surcharge attend en paramètre l'index du noeud fils. La seconde surcharge attend en paramètre le noeud à déconnecter. Dans tout les cas, il est nécessaire d'utiliser cette méthode sur le noeud père.

La méthode removeFromParent() appelée à partir d'un noeud permet de supprimer le lien entre le noeud et son père.

La méthode setParent() permet de préciser le père du noeud.

La méthode setUserObject() permet d'associer un objet au noeud. L'appel à la méthode toString() de cet objet permettra de déterminer la libellé du noeud qui sera affiché.

15.8.2.3. La classe DefaultMutableTreeNode

Généralement, les noeuds créés dans le modèle sont des instances de la classe DefaultMutableTreeNode. Cette classe implémente l'interface MutableTreeNode ce qui permet d'obtenir une instance d'un noeud modifiable.

Généralement, les noeuds fournis en paramètres des méthodes proposées par Swing sont de type TreeNode. Si l'instance du noeud est de type DefaultTreeNode, il est possible de faire un cast pour accéder à toutes ces méthodes.

La classe propose trois constructeurs dont deux attendent en paramètre l'objet qui sera associé au noeud. L'un des deux attend en plus un booléen qui permet de préciser si le noeud peut avoir des noeuds fils.

Constructeur	Rôle
public DefaultMutableTreeNode()	créé un noeud sans objet associé. Cette association pourra être faite avec la méthode setObject()
public DefaultMutableTreeNode(Object userObject)	créé un noeud en précisant l'objet qui lui sera associé et qui pourra avoir des noeuds fils
public DefaultMutableTreeNode(Object userObject, boolean allowsChildren)	créé un noeud dont le booléen précise si il pourra avoir des fils

Pour ajouter une instance de la classe DefaultMutableTreeNode dans le modèle de l'arbre, il est possible d'utiliser la méthode insert() de l'interface MutuableTreeNode ou utiliser la méthode add() de la classe DefaultMutableTreeNode. Celle-ci attend en paramètre une instance du noeud fils à ajouter. Elle ajoute le noeud après le dernier noeud fils, ce qui évite d'avoir à garder une référence sur la position où insérer le noeud.

Exemple (code Java 1.1) :

```
DefaultMutableTreeNode racineNode = new DefaultMutableTreeNode();
DefaultMutableTreeNode division1 = new DefaultMutableTreeNode("Division 1");
DefaultMutableTreeNode division2 = new DefaultMutableTreeNode("Division 2");
racineNode.add(division1);
racineNode.add(division2);
jTree.setModel(new DefaultTreeModel(racineNode));
```

Il est aussi possible de définir sa propre classe qui implémente l'interface MutableTreeNode : une possibilité est de définir une classe fille de la classe DefaultMutableTreeNode.

15.8.3. La modification du contenu de l'arbre

Les modifications du contenu de l'arbre peuvent se faire au niveau du modèle (DefaultTreeModel) ou au niveau du noeud.

La méthode getModel() de la classe JTree permet d'obtenir une référence sur l'instance de la classe TreeModel qui encapsule le modèle de données.

Il est ainsi possible d'accéder à tous les noeuds du modèle pour les modifier.

Exemple (code Java 1.1) :

```
jTree = new JTree();
Object noeudRacine = jTree.getModel().getRoot();
((DefaultMutableTreeNode)noeudRacine).setUserObject("Racine de l'arbre");
```

L'interface TreeModel ne propose rien pour permettre la mise à jour du modèle. Pour mettre à jour le modèle, il faut utiliser une instance de la classe DefaultTreeModel.

Elle propose plusieurs méthodes pour ajouter ou supprimer un noeud :

void insertNodeInto(MutableTreeNode child, MutableTreeNode parent, int index)
void removeNodeFromParent(MutableTreeNode parent)

L'avantage de ces deux méthodes est qu'elles mettent à jour le modèle mais aussi elles mettent à jour la vue en appelant respectivement les méthodes nodesWereInserted() et nodesWereRemoved() de la classe DefaultTreeModel.

Ces deux méthodes sont donc pratiques pour faire des mises à jour mineures mais elles sont peut adaptées pour de nombreuses mises à jour puisque qu'elles lèvent un événement à chacune de leur utilisation.

15.8.3.1. Les modifications des noeuds fils

La classe DefaultMutableTreeNode propose plusieurs méthodes pour mettre à jour le modèle à partir du noeud qu'elle encapsule.

void add(MutableTreeNode child)
void insert(MutableTreeNode child, int index)
void remove(int index)
void remove(MutableTreeNode child)
void removeAllChildren()
void removeFromParent()

Toutes ces méthodes sauf la dernière agissent sur un ou plusieurs noeuds fils. Ces méthodes agissent simplement sur la structure du modèle. Elles ne provoquent pas un affichage par la partie vue de ces changements dans le modèle. Il est nécessaire d'utiliser une des méthodes suivantes proposées par la classe DefaultTreeModel :

Méthode	Rôle
void reload()	rafraichir toute l'arborescence à partir du modèle

void reload(TreeNode node)	rafraichir toute l'arborescence à partir du noeud précisé en paramètre
void nodesWereInserted(TreeNode node, int[] childIndices)	cette méthode rafraichit pour le noeud précisé les noeuds fils ajoutés dont les index sont fournis en paramètre
void nodesWereRemoved(TreeNode node,int[] childIndices, Object[] removedChildren)	cette méthode rafraichit pour le noeud précisé les noeuds fils supprimés dont les index sont fournis en paramètre
void nodeStructureChanged(TreeNode node)	cette méthode est identique à la méthode reload()

15.8.3.2. Les événements émis par le modèle

Il est possible d'enregistrer un listener de type TreeModelListener sur un objet de type DefaultTreeModel.

L'interface TreeModelListener définit quatre méthodes pour répondre à des événements particuliers :

Méthode	Rôle
void treeNodesChanged(TreeModelEvent)	la méthode nodeChanged() ou nodesChanged() est utilisée
void treeStructureChanged(TreeModelEvent)	la méthode reload() ou nodeStructureChanged() est utilisée
void treeNodesInserted(TreeModelEvent)	la méthode nodeWhereInserted() est utilisée
void treeNodesRemoved(TreeModelEvent)	la méthode nodeWhereRemoved() est utilisée

Toutes ces méthodes ont un objet de type TreeModelEvent qui encapsule l'événement.

La classe TreeModelEvent encapsule l'événement et propose cinq méthodes pour obtenir des informations sur les noeuds impactés par l'événement.

Méthode	Rôle
Object getSource()	renvoie une instance sur le modèle de l'arbre (généralement un objet de type DefaultTreeModel)
TreePath getTreePath()	renvoie le chemin du noeud affecté par l'événement
Object[] getPath()	renvoie le chemin du noeud affecté par l'événement
Object[] getChildren()	
int[] getChildIndices()	

Dans la méthode treeStructureChanged(), seules les méthodes getPath() et getTreePath() fournissent des informations utiles en retournant le noeud qui a été modifié.

Dans la méthode treeNodesChanged(), treeNodesRemoved() et treeNodesInserted() les méthodes getPath() et getTreePath() renvoient le noeud père des noeuds affectés. Les méthodes getChildIndices() et getChidren() renvoie respectivement un tableau des index des noeuds fils modifiés et un tableau de ces noeuds fils.

Dans ces méthodes, les méthodes getPath() et getTreePath() renvoient le noeud père des noeuds affectés.

Comme l'objet JTree enregistre ces propres listeners, il n'est pas nécessaire la plupart du temps, d'enregistrer ces listeners hormis pour des besoins spécifiques.

15.8.3.3. L'édition d'un noeud

Par défaut, le composant JTree est readonly. Il est possible d'autoriser l'utilisateur à modifier le libellé des noeuds en utilisant la méthode setEditable() avec le paramètre true. jTtree.setEditable(true);

Pour éditer un noeud, il faut

- sur un noeud non sélectionné : cliquer rapidement trois fois sur le noeud à modifier
- sur un noeud déjà sélectionné : cliquer une fois sur le noeud ou appuyer sur la touche F2

Pour valider les modifications, il suffit d'appuyer sur la touche « Entree ».

Pour annuler les modifications, il suffit d'appuyer sur la touche « Esc »

Il est possible d'enregistrer un listener de type TreeModelListerner pour assurer des traitements lors d'événements liés à l'édition d'un noeud.

L'interface TreeModelListener définit la méthode treeNodesChanged() qui permet de traiter les événements de type TreeModelEvent liés à la modification d'un noeud.

Exemple (code Java 1.1) :

```
jTree.setEditable(true);
jTree.getModel().addTreeModelListener(new TreeModelListener() {

  public void treeNodesChanged(TreeModelEvent evt) {
    System.out.println("TreeNodesChanged");
    Object[] noeuds = evt.getChildren();
    int[] indices = evt.getChildIndices();
    for (int i = 0; i < noeuds.length; i++) {
      System.out.println("Index " + indices[i] + ", nouvelle valeur : "
          + noeuds[i]);
    }
  }

  public void treeStructureChanged(TreeModelEvent evt) {
    System.out.println("TreeStructureChanged");
  }

  public void treeNodesInserted(TreeModelEvent evt) {
    System.out.println("TreeNodesInserted");
  }

  public void treeNodesRemoved(TreeModelEvent evt) {
    System.out.println("TreeNodesRemoved");
  }

});
```

15.8.3.4. Les éditeurs personnalisés

Il est possible de définir un éditeur particulier pour éditer la valeur d'un noeud. Un éditeur particulier doit implémenter l'interface TreeCellEditor.

Cette interface hérite de l'interface CellEditor qui définie plusieurs méthodes utiles pour la définition d'un éditeur dédié :

Object getCellEditorValue();
boolean isCellEditable(EventObject);
boolean shouldSelectCell(EventObject);
boolean stopCellEditing();
void cancelCellEditing();
void addCellEditorListener(CellEditorListener);
void removeCellEditorListener(CellEditorListener);

L'interface TreeCellEditor ne définie qu'une seule méthode :

Component getTreeCellEditorComponent(JTree tree, Object value, boolean isSelected, boolean expanded, boolean leaf, int row);

Cette méthode renvoie un composant qui va permettre l'éditeur de la valeur du noeud.

La valeur initiale est fournie dans le second paramètre de type Object. Les trois arguments de type booléen suivants permettent respectivement de savoir si le noeud est sélectionné, est étendu et est une feuille.

Swing propose une implémentation de cette interface dans la classe DefaultCellEditor qui permet de modifier la valeur du noeud sous la forme d'une zone de texte, d'une case à cocher ou d'une liste déroulante grâce à trois constructeurs :

public DefaultCellEditor(JTextField text); public DefaultCellEditor(JCheckBox box); public DefaultCellEditor(JComboBox combo);

La méthode setCellEditor() de la classe JTree permet d'associer le nouvel éditeur à l'arbre.

Exemple (code Java 1.1) :

```
jTree.setEditable(true);
String[] elements = { "Element 1", "Element 2", "Element 3", "Element  4"};
JComboBox jCombo = new JComboBox(elements);
DefaultTreeCellEditor editor = new DefaultTreeCellEditor(jTree,
         new DefaultTreeCellRenderer(), new DefaultCellEditor(jCombo));
jTree.setCellEditor(editor);
```

15.8.3.5. 3.5 La définition des noeuds éditables

Par défaut, si la méthode setEditable(true) est utilisée alors tous les noeuds sont modifiables.

Il est possible de définir quels sont les noeuds de l'arbre qui sont éditables en créant une classe fille de la classe JTree et en redéfinissant la méthode isPathEditable().

Cette méthode est appelée avant chaque édition d'un noeud. Elle attend en paramètre un objet de type TreePath qui encapsule le chemin du noeud à éditer.

Par défaut, elle renvoie le résultat de l'appel à la méthode isEditable(). Il est d'ailleurs important, lors de la redéfinition de la méthode isPathEditable(), de tenir compte du résultat de la méthode isEditable() pour s'assurer que l'arbre est modifiable avant vérifier si le noeud peut être modifié.

15.8.4. La mise en oeuvre d'actions sur l'arbre

15.8.4.1. Etendre ou refermer un noeud

Pour étendre un noeud et ainsi voir ces fils, l'utilisateur peut double cliquer sur l'icône ou sur le libellé du noeud. Il peut aussi cliquer sur le petit commutateur à gauche de l'icône.

Enfin, il est possible d'utiliser le clavier pour naviguer dans l'arbre à l'aide des touches flèches haut et bas et des touches flèches droite et gauche pour respectivement d'étendre ou de refermer un noeud. Lors d'un appui sur la flèche gauche et que le noeud est déjà fermé alors c'est le noeud père qui est sélectionné. De la même façon, lors d'un appui sur la flèche droite et que le noeud est étendu alors le premier noeud fils est sélectionné.

La touche HOME permet de sélectionner le noeud racine. La touche END permet de sélectionner le noeud qui est la dernière feuille du dernier noeud. Les touches PAGEUP et PAGEDOWN permettent de paginer dans les noeuds de l'arbre.

Depuis Java 2 version 1.3, la méthode setToggleClickCount() permet de préciser le nombre de clic nécessaire pour réaliser l'opération pour étendre ou refermer un noeud.

La classe JTree propose plusieurs méthodes liées aux actions permettant d'étendre ou de refermer un noeud.

Méthode	Rôle
public void expandRow (int row)	Etendre le noeud dont l'index est fourni en paramètre
public void collapseRow(int row)	Refermer le noeud dont l'index est fourni en paramètre
public void expandPath(TreePath path)	Etendre le noeud encapsulé dans la classe TreePath fourni en paramètre
public void collapsePath(TreePath path)	Refermer le noeud encapsulé dans la classe TreePath fourni en paramètre
public boolean isExpanded(int row)	Renvoie un booléen qui précise si le noeud dont l'index est fourni en paramètre est étendu
public boolean isCollapsed (int row)	Renvoie un booléen qui précise si le noeud dont l'index est fourni en paramètre est refermé
public boolean isExpanded(TreePath path)	Renvoie un booléen qui précise si le noeud encapsulé dans la classe TreePath fourni en paramètre est étendu
public boolean isCollapsed (TreePath path)	Renvoie un booléen qui précise si le noeud encapsulé dans la classe TreePath fourni en paramètre est refermé

Par défaut, le noeud racine est étendu.

Les méthodes expandRow() et expandPath() ne permettent que d'étendre les noeuds fils direct du noeud sur lesquelles elles sont appliquées. Pour étendre les noeuds sous jacent il est nécessaire d'écrire du code pour réaliser l'opération sur chaque noeud concerné de façon récursive.

Pour refermer tous les noeux et ne laisser que le noeud racine, il faut utiliser la méthode collapseRow() et lui passant 0 comme paramètre puisque le noeud racine est toujours le premier noeud.

Exemple (code Java 1.1) :

```
jTree.collapseRow(0);
```

La classe JTree propose deux méthodes pour forcer un noeud à être visible : scrollPathToVisible() et scrollRowToVisible(). Celles-ci ne peuvent fonctionner que si le composant JTree est inclus dans un conteneur JScrollPane pour permettre au composant de scroller.

Exemple (code Java 1.1) :

```
      jTree.addTreeExpansionListener(new TreeExpansionListener() {
        public void treeExpanded(TreeExpansionEvent evt) {
          System.out.println("treeExpanded : path=" + evt.getPath());
          jTree.scrollPathToVisible(evt.getPath());
        }
```

15.8.4.2. La détermination du noeud sélectionné

Pour déterminer le noeud sélectionné, il suffit d'utiliser la méthode getLastSelectedPathComponent() de la classe JTree et de caster la valeur retournée dans le type du noeud, généralement de type DefaultMutableTreeNode. La méthode getObject() du noeud permet d'obtenir l'objet associé au noeud. Si l'objet associé est simplement une chaîne de caractères ou si la valeur nécessaire est simplement le libellé du noeud, il suffit d'utiliser la méthode toString().

Exemple (code Java 1.1) : un bouton qui précise lors d'un clic le noeud sélectionné

```
  ...
  private JButton getJButton() {
    if (jButton == null) {
      jButton = new JButton();
      jButton.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(java.awt.event.ActionEvent e) {
          System.out.println("actionPerformed()");
          System.out.println("Noeud sélectionné : "
              + jTree.getLastSelectedPathComponent().toString());
        }
      });
    }
```

15.8.4.3. Le parcours des noeuds de l'arbre

Il peut être nécessaire de parcourir tout ou partie des noeuds de l'arbre pour par exemple faire une recherche dans l'arborescence.

Si l'arbre est composé de noeuds de type DefaultMutableTreenode alors l'interface TreeNode propose plusieurs méthodes pour obtenir une énumération des noeuds pour parcourir tout ou partie de l'arborescence dans les deux sens et deux ordres :

Enumeration preorderEnumeration();
Enumeration postorderEnumeration();
Enumeration breadthFirstEnumeration();
Enumeration depthFirstEnumeration();

Dans l'exemple ci-dessous, l'arborescence suivante est utilisée :

Exemple (code Java 1.1) : un bouton qui précise lors d'un clic le noeud sélectionné

```
Enumeration e = ((DefaultMutableTreeNode)jTree.getModel().getRoot()).preorderEnumeration();
while (e.hasMoreElements()) {
   System.out.println(e.nextElement() + " ");
}
```

Résultat :

preorder	postorder	breadthFirst	depthFirst
Racine de l'arbre colors blue violet red yellow sports basketball soccer football hockey food hot dogs pizza ravioli bananas	blue violet red yellow colors basketball soccer football hockey sports hot dogs pizza ravioli bananas food Racine de l'arbre	Racine de l'arbre colors sports food blue violet red yellow basketball soccer football hockey hot dogs pizza ravioli bananas	blue violet red yellow colors basketball soccer football hockey sports hot dogs pizza ravioli bananas food Racine de l'arbre

La méthode pathFromAncestorEnumeration(TreeNode ancestor) renvoie une énumération des noeuds entre le noeud sur lequel la méthode est appelée et le noeud fourni en paramètre. Ainsi le noeud fourni en paramètre doit obligatoirement être un noeud fils direct ou indirect du noeud sur lequel la méthode est appelée. Dans le cas contraire, une exception de type IllegalArgumentException est levée.

15.8.5. La gestion des événements

Il est possible d'attacher des listeners pour répondre aux événements liés à la sélection d'un élément ou l'extension ou la refermeture d'un noeud.

15.8.5.1. La classe TreePath

Durant son utilisation, le composant JTree ne gère pas directement les noeuds du modèle de données. La manipulation de ces noeuds se fait via un index ou une instance de la classe TreePath.

L'utilisation de l'index est assez délicate car seul le noeud racine de l'arbre possède toujours le même index 0. Pour les autres noeuds, la valeur de l'index dépend de l'état étendu/refermée de chaque noeud puisque seuls les noeuds affichés possèdent un index. Il est donc préférable d'utiliser la classe TreePath.

Le modèle de données utilise des noeuds mais l'interface de l'arbre utilise une autre représentation sous la forme de la classe TreePath.

La classe DefaultMutableTreeNode est la représentation physique d'un noeud, la classe TreePath est la représentation logique. Elle encapsule le chemin du noeud dans l'arborescence.

Cette classe contient plusieurs méthodes :

public Object getLastPathComponent();
public Object getPathComponent(int index);
public int getPathCount();
public Object[] getPath();
public TreePath getParentPath();
public TreePath pathByAddingChild(Object child);
public boolean isDescendant(TreePath treePath)

La méthode getPath() renvoie un tableau d'objets contenant chaque noeud qui compose le chemin encapsulé par la classe TreePath.

La méthode getLastPathComponent() renvoie le dernier noeud du chemin. Cette méthode permet d'obtenir à partir l'instance de la classe TreeNode correspondante dans le modèle de données.

La méthode getPathCount() renvoie le nombre de noeud qui compose le chemin.

La méthode getPathComponent() permet de renvoyer le noeud dont l'index dans le chemin est fourni en paramètre. L'élément avec l'index 0 est toujours le noeud racine de l'arbre.

La méthode getParentPath() renvoie une instance de la classe TreePath qui encapsule le chemin vers le noeud père du chemin encapsulé.

La méthode pathByAddingChild() renvoie une instance de la classe TreePath qui encapsule le chemin issu de l'ajout d'un noeud fils fourni en paramètre.

La méthode idDescendant() renvoie un booléen qui précise si le chemin passé en paramètre est un descendant du chemin encapsulé.

La classe TreePath ne permet pas de gérer le contenu de chaque noeud mais uniquement le chemin de ce noeud dans l'arborescence. Pour accéder au noeud à partir de son chemin, il faut utiliser la méthode getLastPathComponent(). Pour obtenir un noeud inclus dans le chemin, il faut utiliser la getPathComponent() ou getPath(). Toutes ces méthodes renvoient un objet ou un tableau de type Object. Il est donc nécessaire de réaliser un cast vers le type de noeud utilisé, généralement de type DefaultMutableTreeNode.

A partir d'un noeud de type DefaultMutableTreeNode, il est possible d'obtenir l'objet TreePath encapsulant le chemin du noeud en utilisant la méthode getPath() pour obtenir un tableau d'objets de type TreeNode et de passer ce tableau au constructeur de la classe TreePath.

Exemple (code Java 1.1) :

```
TreeNode[] chemin = noeud.getPath(); TreePath path = new TreePath(chemin);
```

15.8.5.2. La gestion de la sélection d'un noeud

La gestion de la sélection de noeud dans un composant JTree est déléguée à un modèle de sélection sous la forme d'une classe qui implémente l'interface TreeSelectionModel. Par défaut, le composant JTree utilise une instance de la classe DefaultTreeSelectionModel.

Le modèle de sélection peut être configuré selon trois modes :

- SINGLE_TREE_SELECTION: un seul noeud peut être sélectionné.
- CONTIGUOUS_TREE_SELECTION: plusieurs noeuds peuvent être sélectionnés à condition d'être contigus.
- DISCONTIGUOUS_TREE_SELECTION: plusieurs noeuds peuvent être sélectionnés de façon continue et/ou discontinue (c'est le mode par défaut).

Pour empêcher la sélection d'un noeud dans l'arbre, il faut supprimer son modèle de sélection en passant null à la méthode setSelectionModel().

Exemple (code Java 1.1) :

```
JTree jTree = new JTree()jTree.setSelectionModel(null);
```

La sélection d'un noeud peut être réalisée par l'utilisateur ou par l'application : le modèle de sélection s'assure que celle-ci est réalisée en respectant le mode de sélection du modèle.

L'utilisateur peut utiliser la souris pour sélectionner un noeud ou appuyer sur la touche Espace sur le noeud courant pour le sélectionner. Il est possible de sélectionner plusieurs noeuds en fonction du mode en maintenant la touche CTRL enfoncée. Avec la touche SHIFT, il est possible selon le mode de sélectionner tous les noeuds entre un premier noeud sélectionné et le noeud sélectionné.

La sélection d'un noeud génère un événement de type TreeSelectionEvent.

Le dernier noeud sélectionné peut être obtenu en utilisant les méthodes getLeadSelectionPath() ou getLeadSelectionRow().

Par défaut la sélection d'un noeud entraine l'extension des noeuds ascendants correspondant afin de les rendre visibles. Pour empêcher ce comportement, il faut utiliser la méthode setExpandSelectedPath() en lui fournissant la valeur false en paramètre.

public void setExpandsSelectedPaths(boolean cond);

Les classes DefaultTreeSelectionModel et JTree possèdent plusieurs méthodes pour gérer la sélection de noeuds. Certaines de ces méthodes sont communes à ces deux classes.

Méthode	Rôle
int getSelectionMode()	renvoie le mode de sélection
void setSelectionMode(int mode)	mettre à jour le mode de sélection
Object getLastSelectedPathComponent()	renvoie le premier noeud de la sélection courante ou null si aucun noeud n'est sélectionné JTree uniquement
TreePath getAnchorSelectionPath()	JTree uniquement
void setAnchorSelectionPath(TreePathpath)	JTree uniquement
TreePath getLeadSelectionPath()	
setLeadSelectionPath()	
int getMaxSelectionRow()	Renvoie le plus grand index de la sélection

int getMinSelectionRow()	Renvoie le plus petit index de la sélection
int getSelectionCount()	Renvoie le nombre de noeud inclus dans la sélection
TreePath getSelectionPath()	Renvoie le chemin du premier élément sélectionné
TreePath[] getSelectionPaths()	Renvoie un tableau des chemins des noeuds inclus dans la sélection
int[] getSelectionRows()	Renvoie un tableau des index des noeuds inclus dans la sélection
Boolean isPathSelected (TreePath path)	Renvoie un booléen si le noeud dont le chemin est fourni en paramètre est inclus dans la sélection
Boolean isRowSelected(int row)	Renvoie un booléen si le noeud dont l'index est fourni en paramètre est inclus dans la sélection
boolean isSelectionEmpty()	Renvoie un booléen qui précise si la sélection est vide
void clearSelection()	Vide la sélection
void removeSelectionInterval (int row0, int row1)	Enlève de la sélection les noeuds dans l'intervalle des index fournis en paramètre
void removeSelectionPath(TreePath path)	Enlève de la sélection le noeud dont le chemin est fourni en paramètre
void removeSelectionRow (int row)	Enlève de la sélection le noeud dont l'index est fourni en paramètre JTree uniquement
void removeSelectionRows(int[] rows)	Enlève de la sélection les noeuds dont les index sont fournis en paramètre JTree uniquement
void addSelectionInterval(int row0, int row1)	Ajouter à la sélection les noeuds dont l'intervalle des index est fourni en paramètre
void addSelectionPath(TreePath path)	Ajouter à la sélection le noeud dont le chemin est fourni en paramètre
addSelectionPaths(TreePath[] path)	Ajouter à la sélection les noeuds dont les chemins sont fournis en paramètre
void addSelectionRow(int row)	Ajouter à la sélection le noeud dont l'index est fourni en paramètre
void addSelectionRows(int[] row)	Ajouter à la sélection les noeuds dont les index sont fournis en paramètre
void setSelectionInterval(int row0, int row1)	Définir la sélection avec les noeuds dont les index sont fournis en paramètre JTree uniquement
setSelectionPath(TreePath path)	Définir la sélection avec le noeud dont le chemin est fourni en paramètre
void setSelectionPaths (TreePath[] path)	Définir la sélection avec les noeuds dont les chemins sont fournis en paramètre
void setSelectionRow(int row)	Définir la sélection avec le noeud dont l'index est fourni en paramètre
void setSelectionRows(int[] row)	Définir la sélection avec les noeuds dont les index sont fournis en paramètre JTree uniquement

15.8.5.3. Les événements lies à la sélection de noeuds

Lors de la sélection d'un noeud, un événement de type TreeSelectionEvent est émis. Pour traiter cet événement, le composant doit enregistrer un listener de type TreeSelectionListener.

L'interface TreeSelectionListener définit une seule méthode :

public void valueChanged(TreeSelectionEvent evt)

Exemple (code Java 1.1) :

```
jTree.addTreeSelectionListener(new javax.swing.event.TreeSelectionListener() {
      public void valueChanged(javax.swing.event.TreeSelectionEvent e) {

        DefaultMutableTreeNode noeud = (DefaultMutableTreeNode) jTree
            .getLastSelectedPathComponent();
        if (noeud == null)
          return;
        System.out.println("valueChanged() : " + noeud);
      }
});
```

La classe TreeSelectionEvent possède plusieurs méthodes pour obtenir des informations sur la sélection.

Méthode	Rôle
public TreePath[] getPaths()	renvoie un tableau des chemins des noeuds sélectionnés
public boolean isAddedPath (TreePath path)	Renvoie true si le noeud sélectionné est ajouté à la sélection. Renvoie false si le noeud sélectionné est retiré de la sélection
TreePath getPath()	Renvoie le chemin du premier noeud sélectionné
boolean isAddedPath()	Renvoie true si le premier noeud sélectionné est ajouté à la sélection. Renvoie false si le premier noeud sélectionné est retiré de la sélection
TreePath getOldLeadSelection()	
TreePath getNewLeadSelection()	

Un listener de type TreeSelectionListener est enregistré en utilisant la méthode addTreeSelectionListener()de la classe JTree.

Exemple (code Java 1.1) :

```
jTree.addTreeSelectionListener(new TreeSelectionListener() {

  public void valueChanged(TreeSelectionEvent e) {

    Object obj = jTree.getLastSelectedPathComponent();
    System.out.println("getLastSelectedPathComponent=" + obj);
    System.out.println("getPath=" + e.getPath());
    System.out.println("getNewLeadSelectionPath="
        + e.getNewLeadSelectionPath());
    System.out.println("getOldLeadSelectionPath="
        + e.getOldLeadSelectionPath());
    TreePath[] paths = e.getPaths();

    for (int i = 0; i < paths.length; i++) {
      System.out.println("Path " + i + "=" + paths[i]);
    }
  }
});
```

Un événement de type TreeSelectionEvent n'est émis que si un changement intervient dans la sélection : lors d'un clic sur un noeud, celui est sélectionné et un événement est émis. Lors d'un clic sur ce même noeud, le noeud est toujours sélectionné mais l'événement n'est pas émis puisque la sélection n'est pas modifiée.

Dans un listener pour gérer les événements de la souris, il est possible d'utiliser la méthode getPathForLocation() pour déterminer le chemin d'un noeud à partir des coordonnées de la souris qu'il faut lui fournir en paramètre.

La méthode getPathForLocation() renvoie null si l'utilisateur clic en dehors d'un noeud dans l'arbre.

Exemple (code Java 1.1) :

```
jTree.addMouseListener(new MouseAdapter() {
  public void mouseClicked(MouseEvent evt) {
      TreePath path =
         jTree.getPathForLocation(evt.getX(), evt.getY());
      if (path != null) {
         System.out.println("path= " + path.getLastPathComponent());
       }
  }
});
```

Plusieurs autres méthodes peuvent aussi être utilisées dans ce contexte.

Méthode	Rôle
TreePath getClosestPathForLocation(int x, int y)	Retourne le chemin du noeud le plus proche des coordonnées fournies en paramètre
int getClosestRowForLocation(int x, int y)	Retourne l'index du noeud le plus proche des coordonnées fournies en paramètre
Rectangle getPathBounds(TreePath path)	Renvoie un objet de type Rectangle qui représente la surface du noeud dont le chemin est fourni en paramètre
TreePath getPathForLocation(int x, int y)	Retourne le chemin du noeud dont la surface contient les coordonnées fournies en paramètre. Renvoie null si ces coordonnées ne correspondent à aucune
TreePath getPathForRow(int row)	Renvoie le chemin du noeud dont l'index est fourni en paramètre
Rectangle getRow-Bounds(int row)	Renvoie un objet de type Rectangle qui représente la surface du noeud dont l'index est fourni en paramètre
int getRowForLocation(int x, int y)	

15.8.5.4. Les événements lorsqu'un noeud est étendu ou refermé

A chaque fois qu'un noeud est étendu ou refermé, un événement de type TreeExpansionEvent est émis. Il est possible de répondre à ces événements en mettant en place un listener de type TreeExpansionListener.

L'interface TreeExpansionListener propose deux méthodes :

public void treeExpanded(TreeExpansionEvent event) public void treeCollapsed(TreeExpansionEvent event)

La classe TreeExpansionEvent possède une propriété source qui contient une référence sur le composant JTree à l'origine de l'événement et une propriété path qui contient un objet de type TreePath encapsulant le chemin du noeud à l'origine de l'événement.

Les valeurs de ces deux propriétés peuvent être obtenus avec leur getter respectif : getSource() et getPath().

Exemple (code Java 1.1) :

```
jTree.addTreeExpansionListener(new TreeExpansionListener() {

  public void treeExpanded(TreeExpansionEvent evt) {
    System.out.println("expand, path=" +
                          evt.getPath());
  }

  public void treeCollapsed(TreeExpansionEvent evt) {
    System.out.println("collapse, path=" +
                         evt.getPath());
  }
```

```
});
```

Un seul événement est généré à chaque fois qu'un noeud est étendu ou refermé : il n'y a pas d'événements émis pour les éventuels noeuds fils qui sont étendu ou refermé suite à l'action.

15.8.5.5. Le contrôle des actions pour étendre ou refermer un noeud

Il peut être utile de recevoir un événement avant qu'un noeud ne soit étendu ou refermé. Un listener de type TreeWillExpandListener() peut être mis en place pour recevoir un événement de type TreeExpansionEvent lors d'une tentative pour étendre ou refermé un noeud.

L'interface TreeWillExpandListener définit deux méthodes :

public void treeWillCollapse(TreeExpansionEvent evt) throws ExpandVetoException;
public void treeWillExpand(TreeExpansionEvent evt) throws ExpandVetoException;

Les deux méthodes peuvent lever une exception de type ExpandVetoException. Cette méthode doit lever cette exception dans les traitements de ces méthodes si des conditions sont remplies pour empêcher l'action demandée par l'utilisateur. Si l'exception n'est pas levée à la fin des traitements de la méthode alors l'action est réalisée.

Exemple (code Java 1.1) : empécher tous les noeuds étendus de se refermer

```
jTree.addTreeWillExpandListener(new TreeWillExpandListener() {

  public void treeWillCollapse(TreeExpansionEvent event) throws ExpandVetoException {
    throw new ExpandVetoException(event);
  }

  public void treeWillExpand(TreeExpansionEvent event) throws ExpandVetoException {
  }

});
```

15.8.6. La personnalisation du rendu

Le rendu du composant JTree dépend bien sûre dans un premier temps du look and feel utilisé mais il est aussi possible de personnaliser plus finement le rendu des noeuds du composant.

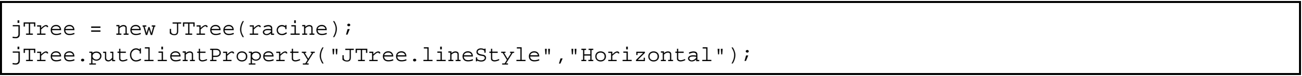
Il est possible de préciser la façon dont les lignes reliant les noeuds sont rendues via une propriété client nommée lineStyle. Cette propriété peut prendre trois valeurs :

Valeur	Rôle
Angled	Une ligne à angle droit relie chaque noeud fils à son noeud père
None	Aucune ligne n'est affichée entre les noeuds
Horizontal	Une simple ligne horizontale sépare les noeuds enfants du noeud racine

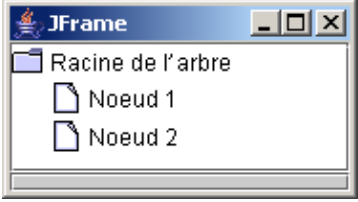
Pour préciser la valeur de la propriété que le composant doit utiliser, il faut utiliser la méthode putClientProperty() qui attend deux paramètres sous forme de chaînes de caractères :

- le nom de la propriété
- sa valeur

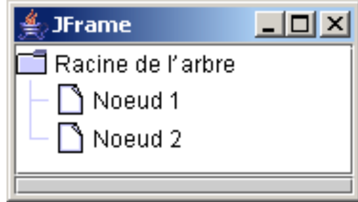
Exemple (code Java 1.1) :



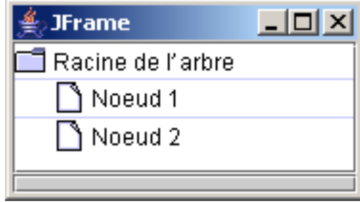
```
jTree = new JTree(racine);
jTree.putClientProperty("JTree.lineStyle","Horizontal");
```



None



Angled



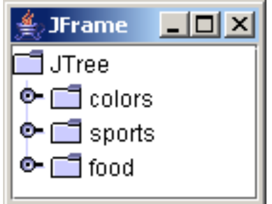
Horizontal

Il est possible de modifier l'apparence de la racine de l'arbre grâce à deux méthodes de la classe JTree : setRootVisible() et setShowsRootHandles().

La méthode setRootVisible() permet de préciser avec son booléen en paramètre si la racine est affichée ou non.

Exemple (code Java 1.1) :

```
JTree jtree = new JTree();
jtree.setShowsRootHandles(false);
jtree.setRootVisible(true);
```



15.8.6.1. Personnaliser le rendu des noeuds

Il est possible d'obtenir un contrôle total sur le rendu de chaque noeud en définissant un objet qui implémente l'interface TreeCellRender. Attention, le rendu personnalisé est parfois dépendant du look & feel utilisé.

L'interface TreeCellRenderer ne définit qu'une seule méthode :

Component getTreeCellRendererComponent(JTree tree, Object value, boolean selected, boolean expanded, boolean leaf, int row, boolean hasFocus)

Cette méthode envoie un composant qui va encapsuler le rendu du noeud. Le premier argument de type JTree encapsule le composant JTree lui-même. L'argument de type Object encapsule le noeud dont le rendu doit être généré.

La méthode getCellRender() renvoie un objet qui encapsule le TreeCellRender. Il est nécessaire de réaliser un cast vers le type de cet objet.

Swing propose une classe de base DefaultTreeCellRender pour le rendu. Elle propose plusieurs méthodes pour permettre de définir le rendu.

Méthode	Rôle
void setBackgroundNonSelectionColor(Color)	Permet de définir la couleur de fond du noeud lorsqu'il n'est pas sélectionné
void setBackgroundSelectionColor(Color)	Permet de définir la couleur de fond du noeud lorsqu'il est sélectionné

void setBorderSelectionColor(Color)	Permet de définir la couleur de la bordure du noeud lorsqu'il est sélectionné. Il n'est pas possible de définir une bordure pour un noeud sélectionné
void setTextNonSelectionColor(Color)	Permet de définir la couleur du texte du noeud lorsqu'il n'est pas sélectionné
void setTextSelectionColor(Color)	Permet de définir la couleur du texte du noeud lorsqu'il est sélectionné
void setFont(Font)	Permet de définir la police de caractère utilisé pour afficher le texte du noeud
void setClosedIcon(Icon)	Permet de définir l'icône associée au noeud lorsque celui-ci est fermé
void setOpenIcon(Icon)	Permet de définir l'icône associée au noeud lorsque celui-ci est étendu
void setLeafIcon(Icon)	Permet de définir l'icône associée au noeud lorsque celui-ci est une feuille

Un composant ne peut avoir qu'une seule instance de type TreeCellRenderer. Cette instance sera donc appelée pour définir le rendu de chaque noeud.

Exemple (code Java 1.1) :

```
TreeCellRenderer cellRenderer = jTree.getCellRenderer();
if (cellRenderer instanceof DefaultTreeCellRenderer) {
  DefaultTreeCellRenderer renderer = (DefaultTreeCellRenderer)cellRenderer;
  renderer.setBackgroundNonSelectionColor(Color.gray);
  renderer.setBackgroundSelectionColor(Color.black);
  renderer.setTextSelectionColor(Color.white);
  renderer.setTextNonSelectionColor(Color.black);
  jTree.setBackground(Color.gray);
}
```

Résultat :

Pour modifier les icônes utiliser par les différents éléments de l'arbre, il faut utiliser les méthodes setOpenIcon(), setClosedIcon() et setLeafIcon().

Méthode	Rôle
setOpenIcon()	précise l'icône pour un noeud ouvert
setClosedIcon()	précise l'icône pour un noeud fermé
setLeafIcon()	précise l'icône pour une feuille

Pour simplement supprimer l'affichage de l'icône, il suffit de passer null à la méthode concernée ou de lui fournir une image sous la forme d'un objet de type

Exemple (code Java 1.1) :

```
DefaultTreeCellRenderer monRenderer = new DefaultTreeCellRenderer();
monRenderer.setOpenIcon(null);
monRenderer.setClosedIcon(null);
monRenderer.setLeafIcon(null);
```

Pour préciser une image, il est faut créer une instance de la classe ImageIcon encapsulant l'image et la passer en paramètre de la méthode concernée.

Exemple (code Java 1.1) :

```
    private Icon ourvertIcon = new ImageIcon("images/ouvert.gif");
    private Icon fermeIcon   = new ImageIcon("images/ferme.gif");
    private Icon feuilleIcon = new ImageIcon("images/feuille.gif");
...
    DefaultTreeCellRenderer treeCellRenderer = new DefaultTreeCellRenderer();
    treeCellRenderer.setOpenIcon(ouvertIcon);
    treeCellRenderer.setClosedIcon(fermeIcon);
    treeCellRenderer.setLeafIcon(feuilleIcon);
```

Il est aussi possible de définir une classe qui hérite de la classe DefaultTreeCellRenderer. Cette classe propose une implémentation par défaut de l'interface TreeCellRenderer. Comme elle hérite de la classe JLabel, elle possède déjà de nombreuses méthodes pour assurer le rendu du noeud sous la forme d'un composant de type étiquette.

Exemple (code Java 1.1) :

```
import java.awt.Color;
import java.awt.Component;

import javax.swing.JTree;
import javax.swing.tree.DefaultTreeCellRenderer;

public class MonTreeCellRenderer extends DefaultTreeCellRenderer {

  public Component getTreeCellRendererComponent(JTree tree, Object value,
      boolean selected, boolean expanded, boolean leaf, int row,
      boolean hasFocus) {


    super.getTreeCellRendererComponent(tree,value, selected, expanded,
        leaf, row,hasFocus);

    setBackgroundNonSelectionColor(Color.gray);
    setBackgroundSelectionColor(Color.black);
    setTextSelectionColor(Color.white);
    setTextNonSelectionColor(Color.black);

    return this;
  }
}
```

Une fois la classe de type DefaultTreeCellRenderer instanciée, il faut utiliser la méthode setCellRenderer() de la classe JTree pour indiquer à l'arbre d'utiliser cette classe pour le rendu.

Exemple (code Java 1.1) :

```
    jTree.setCellRenderer(new MonTreeCellRenderer());
```

La création d'une classe fille de la classe DefaultTreeCellRenderer ne fonctionne correctement qu'avec les look and feel Metal et Windows car le look and feel Motif définit son propre Renderer.

15.8.6.2. Les bulles d'aides (Tooltips)

Le composant JTree ne propose par de support pour les bulles d'aide en standard. Pour permettre à un composant JTree d'afficher une bulle d'aide, il faut :

- enregistrer le composant JTree auprès du ToolTipManager
- définir le contenu de la bulle d'aide dans le Renderer

L'enregistrement du composant auprès du ToolTipManager se fait en utilisant la méthode registerComponent() sur l'instance partagée.

Exemple (code Java 1.1) :

```
ToolTipManager.sharedInstance().registerComponent(jTree);
((JLabel)t.getCellRenderer()).setToolTipText("Arborescence des données");
```

L'inconvénient de cette méthode est que la bulle d'aide est toujours la même quelque soit la position de la souris sur tous les noeuds du composant. Pour assigner une bulle d'aide particulière à chaque noeud, il est nécessaire d'utiliser la méthode setToolTipText() dans la méthode getTreeCellRendererComponent() d'une instance fille de la classe DefaultTreeCellRenderer

Exemple (code Java 1.1) :

```
      jTree.setCellRenderer(new DefaultTreeCellRenderer() {

        public Component getTreeCellRendererComponent(JTree tree, Object value,
            boolean selected, boolean expanded, boolean leaf, int row,
            boolean hasFocus) {

          super.getTreeCellRendererComponent(tree, value, selected, expanded, leaf, row,
              hasFocus);
          setToolTipText(value.toString());

          return this;
        }
      });

      ToolTipManager.sharedInstance().registerComponent(jTree);
```

15.9. Les menus

Les menus de Swing proposent certaines caractéristiques intéressantes en plus de celle proposées par un menu standard :

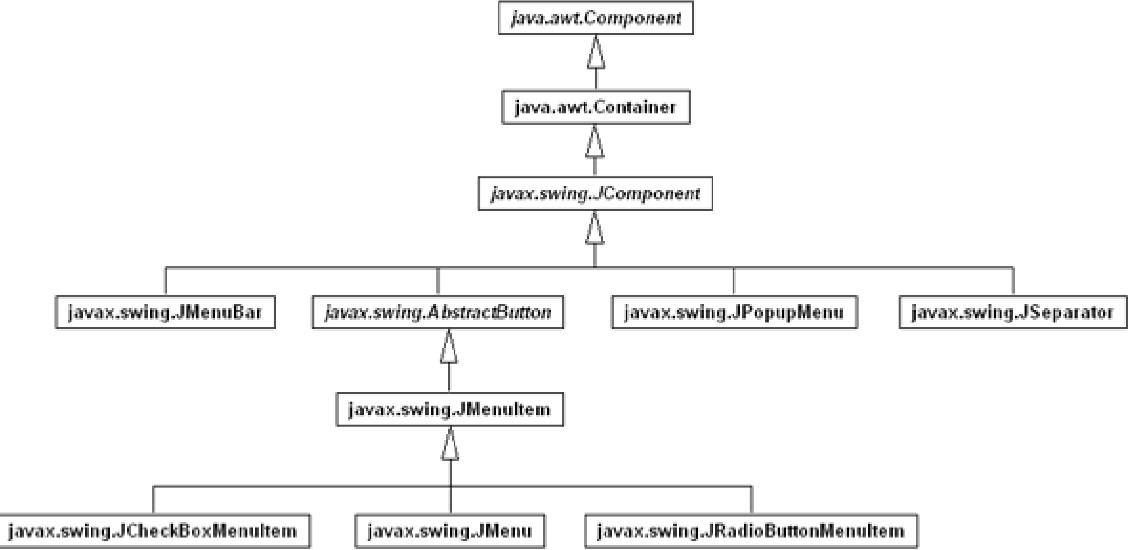
- les éléments de menu peuvent contenir une icône
- les éléments de menu peuvent être de type bouton radio ou case à cocher
- les éléments de menu peuvent avoir des raccourcis clavier (accelerators)

Les menus sont mis en oeuvre dans Swing avec un ensemble de classe :

- JMenuBar : encapsule une barre de menu
- JMenu : encapsule un menu
- JMenuItem : encapsule un élément d'un menu
- JCheckBoxMenuItem : encapsule un élément d'un menu sous la forme d'une case à cocher
- JRadioButtonMenuItem : encapsule un élément d'un menu sous la forme d'un bouton radio
- JSeparator : encapsule un élément d'un menu sous la forme d'un séparateur

* JPopupMenu : encapsule un menu contextuel

Toutes ces classes héritent de façon directe ou indirecte de la classe JComponent.



Les éléments de menus cliquables héritent de la classe JAbstractButton.

JMenu hérite de la classe JMenuItem et non pas l'inverse car chaque JMenu contient un JMenuItem implicite qui encapsule le titre du menu.

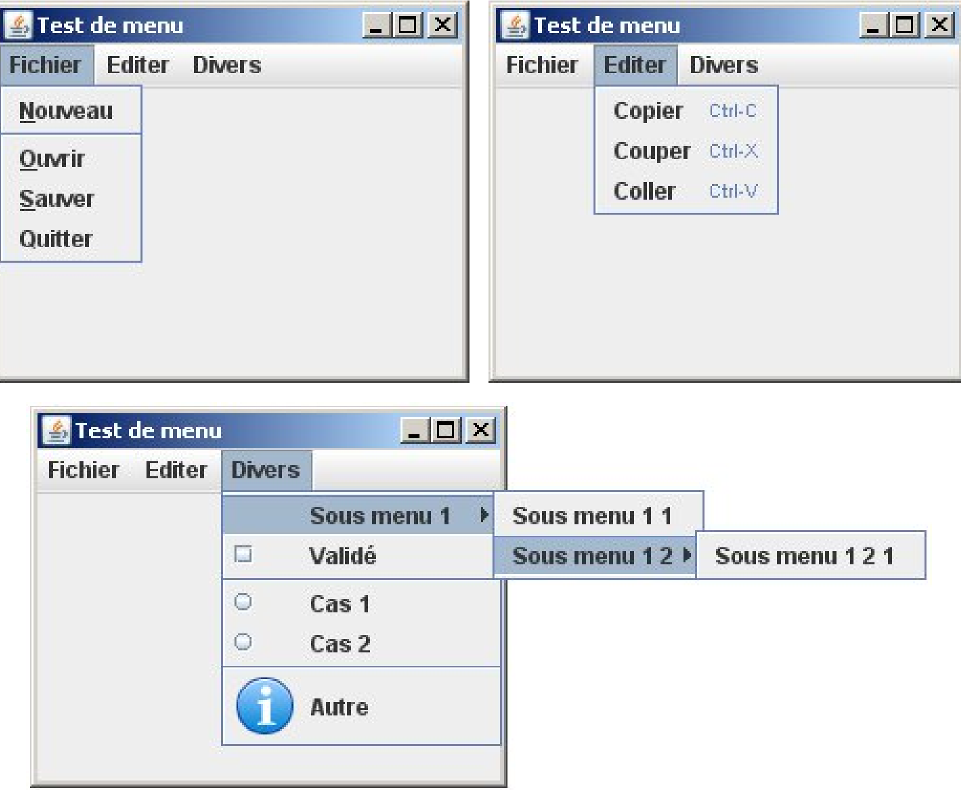
La plupart des classes utilisées pour les menus implémentent l'interface MenuElement. Cette interface définit des méthodes pour la gestion des actions standards de l'utilisateur. Ces actions sont gérées par la classe MenuSelectionManager.

Exemple :

```
package com.jmdoudoux.test.swing.menu;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class TestMenuSwing1 extends JMenuBar {

  public TestMenuSwing1() {

    // Listener générique qui affiche l'action du menu utilisé
    ActionListener afficherMenuListener = new ActionListener() {
      public void actionPerformed(ActionEvent event) {
        System.out.println("Elément de menu [" + event.getActionCommand()
            + "] utilisé.");
      }
    };

    // Création du menu Fichier
    JMenu fichierMenu = new JMenu("Fichier");
    JMenuItem item = new JMenuItem("Nouveau", 'N');
    item.addActionListener(afficherMenuListener);
    fichierMenu.add(item);
    item = new JMenuItem("Ouvrir", 'O');
    item.addActionListener(afficherMenuListener);
    fichierMenu.add(item);
    item = new JMenuItem("Sauver", 'S');
    item.addActionListener(afficherMenuListener);
    fichierMenu.insertSeparator(1);
    fichierMenu.add(item);
    item = new JMenuItem("Quitter");
    item.addActionListener(afficherMenuListener);
    fichierMenu.add(item);
```

```
    // Création du menu Editer
    JMenu editerMenu = new JMenu("Editer");
    item = new JMenuItem("Copier");
    item.addActionListener(afficherMenuListener);
    item.setAccelerator(KeyStroke.getKeyStroke('C', Toolkit.getDefaultToolkit()
        .getMenuShortcutKeyMask(), false));
    editerMenu.add(item);
    item = new JMenuItem("Couper");
    item.addActionListener(afficherMenuListener);
    item.setAccelerator(KeyStroke.getKeyStroke('X', Toolkit.getDefaultToolkit()
        .getMenuShortcutKeyMask(), false));
    editerMenu.add(item);
    item = new JMenuItem("Coller");
    item.addActionListener(afficherMenuListener);
    item.setAccelerator(KeyStroke.getKeyStroke('V', Toolkit.getDefaultToolkit()
        .getMenuShortcutKeyMask(), false));
    editerMenu.add(item);

    // Création du menu Divers
    JMenu diversMenu = new JMenu("Divers");
    JMenu sousMenuDiver1 = new JMenu("Sous menu 1");

    item.addActionListener(afficherMenuListener);
    item = new JMenuItem("Sous menu 1 1");
    sousMenuDiver1.add(item);
    item.addActionListener(afficherMenuListener);
    JMenu sousMenuDivers2 = new JMenu("Sous menu 1 2");
    item = new JMenuItem("Sous menu 1 2 1");
    sousMenuDivers2.add(item);
    sousMenuDiver1.add(sousMenuDivers2);

    diversMenu.add(sousMenuDiver1);
    item = new JCheckBoxMenuItem("Validé");
    diversMenu.add(item);
    item.addActionListener(afficherMenuListener);
    diversMenu.addSeparator();
    ButtonGroup buttonGroup = new ButtonGroup();
    item = new JRadioButtonMenuItem("Cas 1");
    diversMenu.add(item);
    item.addActionListener(afficherMenuListener);
    buttonGroup.add(item);
    item = new JRadioButtonMenuItem("Cas 2");
    diversMenu.add(item);
    item.addActionListener(afficherMenuListener);
    buttonGroup.add(item);
    diversMenu.addSeparator();
    diversMenu.add(item = new JMenuItem("Autre",
        new ImageIcon("about_32.png")));
    item.addActionListener(afficherMenuListener);

    // ajout des menus à la barre de menu
    add(fichierMenu);
    add(editerMenu);
    add(diversMenu);
  }

  public static void main(String s[]) {
    JFrame frame = new JFrame("Test de menu");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setJMenuBar(new TestMenuSwing1());
    frame.setMinimumSize(new Dimension(250, 200));
    frame.pack();
    frame.setVisible(true);
  }
}
```

Résultat :



15.9.1. La classe JMenuBar

La classe JMenuBar encapsule une barre de menu qui contient zéro ou plusieurs menus.

La classe JMenuBar utilise la classe DefaultSingleSelectionModel comme modèle de données : un seul de ces menus peut être activé à un instant T.

Pour ajouter des menus à la barre de menu, il faut utiliser la méthode add() de la classe JMenuBar qui attend en paramètre l'instance du menu.

Pour ajouter la barre de menu à une fenêtre, il faut utiliser la méthode setJMenuBar() d'une instance des classes JFrame, JInternalFrame, JDialog ou JApplet.

Comme la classe JMenuBar hérite de la classe JComponent, il est aussi possible d'instancier plusieurs JMenuBar et de les insérer dans un gestionnaire de positionnement comme n'importe quel composant. Ceci permet aussi de placer le menu à sa guise.

Exemple :

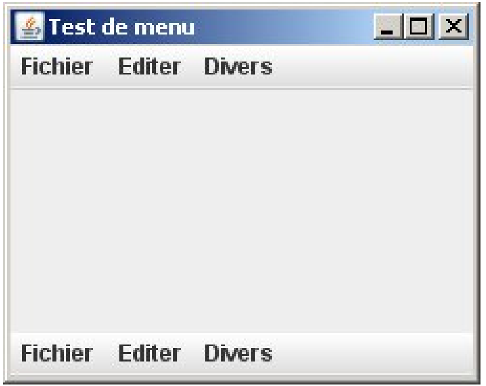
```
...
  public static void main(String s[]) {
    JFrame frame = new JFrame("Test de menu");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    TestMenuSwing1 menu = new TestMenuSwing1();
    frame.getContentPane().add(menu, BorderLayout.SOUTH);
    frame.setMinimumSize(new Dimension(250, 200));
    frame.pack();
    frame.setVisible(true);
  }
...
```

Résultat :

Swing n'impose pas d'avoir un unique menu par fenêtre : il est possible d'avoir plusieurs menus dans une même fenêtre.

Exemple :

```
...
  public static void main(String s[]) {
    JFrame frame = new JFrame("Test de menu");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setJMenuBar(new TestMenuSwing1());
    TestMenuSwing1 menu = new TestMenuSwing1();
    frame.getContentPane().add(menu, BorderLayout.SOUTH);
    frame.setMinimumSize(new Dimension(250, 200));
    frame.pack();
    frame.setVisible(true);
  }
...
```



La classe JMenuBar ne possède qu'un seul constructeur sans paramètre.

Les principales méthodes de la classe JMenuBar sont :

Méthodes	Rôle
JMenu add(JMenu)	Ajouter un menu à la barre de menu
JMenu getMenu(int)	Obtenir le menu dont l'index est fourni en paramètre
int getMenuCount()	Obtenir le nombre de menu de la barre de menu
MenuElement[] getSubElements()	Obtenir un tableau de tous les menus
boolean isSelected()	précise si un menu est affiché
void setMenuHelp (JMenu)	Cette méthode n'est pas implémentée et lève systématiquement une exception

15.9.2. La classe JMenuItem

La classe JMenuItem encapsule les données d'un élément de menu (libellé et/ou image). Elle hérite de la classe AbstractButton. Le comportement est similaire mais différent de celui d'un bouton : avec la classe JMenuItem, le composant est considéré comme sélectionné dès que le curseur de la souris passe dessus.

Les éléments de menu peuvent être associés à deux types de raccourcis clavier :

- les accelerators : ils sont hérités de JComponent : ce sont des touches (par exemple les touches de fonctions) ou des combinaisons de touches avec les touches shift, Ctrl ou Alt qui sont affichées à la droite du libellé de l'élément du menu
- les mnemonics : ils apparaissent sous la forme d'une lettre soulignée. Ils sont utilisables seulement sur certaines plate-formes (par exemple en combinaison avec la touche Alt sous Windows).

La méthode setAccelerator() permet d'associer un accelerator à un élément de type JMenuItem.

Un mnemonic peut être associé à JMenuItm de deux façons :

- soit dans la surcharge du constructeur prévu à cet effet
- soit en utilisant la méthode setMnemonic()

Le mnemonic correspond à un caractère qui doit obligatoirement être contenu dans le libellé.

Un élément de menu peut contenir uniquement une image ou être composé d'un libellé et d'une image. Une image peut être associée à un JMenuItem de deux façons :

- soit dans une des surcharges du constructeur prévu à cet effet
  item = new JMenuItem("Autre", new ImageIcon("about_32.png"));
  item = new JMenuItem(new ImageIcon("about_32.png"));

- soit en utilisant la méthode setIcon
  item.setIcon(new ImageIcon("about_32.png"));

15.9.3. La classe JPopupMenu

La classe JPopupMenu encapsule un menu flottant qui n'est pas rattaché à une barre de menu mais à un composant.

La création d'un JPopMenu est similaire à la création d'un JMenu.

Il est préférable d'ajouter un élément de type JMenuItem grâce à la méthode add() de la classe JPopupMenu mais il est aussi d'ajouter n'importe quel élément qui hérite de la classe Component en utilisant une surcharge de la méthode add().

Il est possible d'ajouter un élément à un index précis en utilisant la méthode insert().

La méthode addSeparator() permet d'ajouter un élément séparateur.

Pour afficher un menu flottant, il faut ajouter un listener sur l'événement déclenchant et utiliser la méthode show() de la classe JPopupMenu.

Exemple :

```
package com.jmdoudoux.test.swing.menu;

import java.awt.Dimension;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;

import javax.swing.JFrame;
import javax.swing.JMenuBar;
```

```
import javax.swing.JMenuItem;
import javax.swing.JPopupMenu;
import javax.swing.JTextField;

public class TestMenuSwing2 extends JMenuBar {

  public JPopupMenu popup;

  public TestMenuSwing2() {

    JMenuItem item = null;

    // Listener générique qui affiche l'action du menu utilisé
    ActionListener afficherMenuListener = new ActionListener() {
      public void actionPerformed(ActionEvent event) {
        System.out.println("Elément de menu [" + event.getActionCommand()
            + "] utilisé.");
      }
    };

    popup = new JPopupMenu();
    item = new JMenuItem("Copier");
    item.addActionListener(afficherMenuListener);
    popup.add(item);
    item = new JMenuItem("Couper");
    item.addActionListener(afficherMenuListener);
    popup.add(item);

  }

  public void processMouseEvent(MouseEvent e) {
  }

  public static void main(String s[]) {
    final JFrame frame = new JFrame("Test de menu divers");
    final JTextField texte = new JTextField();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    final TestMenuSwing2 tms = new TestMenuSwing2();
    frame.add(texte);

    texte.addMouseListener(new MouseAdapter() {

      public void mouseClicked(MouseEvent e) {
        System.out.println("mouse clicked");
        afficherPopup(e);
      }

      public void mousePressed(MouseEvent e) {
        System.out.println("mouse pressed");
        afficherPopup(e);
      }

      public void mouseReleased(MouseEvent e) {
        System.out.println("mouse released");
        afficherPopup(e);
      }

      private void afficherPopup(MouseEvent e) {
        if (e.isPopupTrigger()) {
          tms.popup.show(texte, e.getX(), e.getY());
        }
      }
    });

    frame.setMinimumSize(new Dimension(250, 200));
    frame.pack();
    frame.setVisible(true);
  }
}
```

Le plus simple pour être multiplateforme est de tester sur tous les événements de la souris si cet événement permet l'affichage du menu flottant. Ce test est réalisé grâce à la méthode isPopupTrigger() de la classe MouseEvent.

La propriété invoker encapsule le composant à l'origine de l'affichage du menu déroulant.
La propriété borderPaint indique si la bordure du menu déroulant doit être dessinée.
La propriété visible indique si le menu déroulant est affiché.
La propriété location indique les coordonnées d'affichage du menu déroulant

Un objet de type JPopupMenu peut émettre un événement de type PopupMenuEvent. Ceux-ci sont traités par un listener de type PopupMenuListener qui définit trois méthodes :

Méthode	Rôle
popupMenuCanceled()	méthode appelée avant que l'affichage du menu déroulant ne soit annulé
popoupMenuWillBecomeInvisible()	méthode appelée avant que le menu déroulant ne devienne invisible
popoupMenuWillBecomeVisible()	méthode appelée avant que le menu déroulant ne devienne visible. Cette méthode permet de personnaliser l'affichage des éléments du menu en fonction du contexte (exemple : rendre actif ou non certains élément du menu)

15.9.4. La classe JMenu

La classe JMenu encapsule un menu qui est attaché à un objet de type JMenuBar ou à un autre objet de type JMenu. Dans ce second cas, l'objet est un sous menu.

Il est possible d'ajouter un élément sous la forme d'un objet de type JMenuItem, Component ou Action en utilisant la méthode add(). Chaque élément du menu possède un index.

La méthode addSeparator() permet d'ajouter un élément de type séparateur.
La méthode remove() permet de supprimer un élément du menu en fournissant en paramètre l'instance de l'élément ou son index. Si la suppression réussie, les index des éléments suivants sont décrémentés d'une unité.

La classe JMenu possède plusieurs propriétés :

Propriété	Rôle
popupMenu	JPopupMenu qui encapsule les éléments du menu
topLevelMenu	propriété en lecture seule qui précise si le menu est attaché à un JMenuBar. La valeur false indique que le menu est un sous menu attaché à un autre menu
itemCount	indique le nombre d'éléments du menu (incluant les séparateurs)
delay	précise le temps en milliseconde avant l'affichage du menu
menuComponentCount	indique le nombre de composants du menu
tearOff	ne pas utiliser cette propriété qui lève une exception de type Error

La méthode getMenuComponent() permet d'obtenir le composant du menu dont l'index est fourni en paramètre. La méthode getItem() permet d'obtenir le JMenuItem dont l'index est fourni en paramètre.

La méthode menuComponents() renvoie un tableau des composants du menu.

La méthode isMenuComponent() renvoie un booléen qui précise si le composant fourni en paramètre est inclus dans les éléments du menu.

Un événement de type MenuEvent est émis lorsque le titre du menu est cliqué. Un listener de type MenuListener permet de s'abonner à ces événements. L'interface MenuListener définie trois méthodes qui possède un paramètre de type MenuEvent :

Méthodes	Rôle
menuCanceled()	invoquée lorsque le menu est effacé

menuDeselected()	invoquée lorsque le titre du menu est désélectionné
menuSelected()	invoquée lorsque le titre du menu est sélectionné

15.9.5. La classe JCheckBoxMenuItem

Cette classe encapsule un élément du menu qui contient une case à cocher.

Elle possède de nombreux constructeurs qui permettent de préciser le texte, une icône et l'état de la case à cocher.

La propriété state() permet de connaître ou de positionner l'état de la case à cocher.

15.9.6. La classe JRadioButtonMenuItem

Cette classe encapsule un élément de menu qui contient un bouton radio. Les boutons radio associés à un même groupe sont mutuellement non sélectionnable (un seul bouton radio du groupe peut être sélectionné à la fois).

La définition de ce groupe ce fait en utilisant la classe ButtonGroup. C'est d'ailleurs cette classe qui propose la méthode getSelected() pour connaître le bouton radio sélectionné dans le groupe.

Exemple :

```
...
        diversMenu.addSeparator();
    ButtonGroup buttonGroup = new ButtonGroup();
    item = new JRadioButtonMenuItem("Cas 1");
    diversMenu.add(item);
    item.addActionListener(afficherMenuListener);
    buttonGroup.add(item);
    item = new JRadioButtonMenuItem("Cas 2");
    diversMenu.add(item);
    item.addActionListener(afficherMenuListener);
    buttonGroup.add(item);
    diversMenu.addSeparator();
...
```

15.9.7. La classe JSeparator

La méthode addSeparator() des classe JMenu et JPopupMenu instancie un objet de type JSeparator et l'ajoute à la liste des éléments du menu.

La classe JSeparator encapsule un séparateur dans un menu.

Remarque : cette classe n'est pas utilisable que dans un menu mais peut aussi être utilisée comme un composant dans l'interface.

Exemple :

```
package com.jmdoudoux.test.swing.menu;

import java.awt.Dimension;

import javax.swing.Box;
import javax.swing.BoxLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JSeparator;
import javax.swing.JTextField;
```

```
public class TestMenuSwing3 extends JPanel {

  public TestMenuSwing3() {
    super(true);

    setLayout(new BoxLayout(this, BoxLayout.Y_AXIS));
    Box box1 = new Box(BoxLayout.X_AXIS);
    Box box2 = new Box(BoxLayout.X_AXIS);
    Box box3 = new Box(BoxLayout.X_AXIS);
    Box box4 = new Box(BoxLayout.X_AXIS);
    Box box5 = new Box(BoxLayout.X_AXIS);
    box1.add(new JButton("Bouton 1"));
    box1.add(new JButton("Bouton 2"));
    box1.add(new JButton("Bouton 3"));
    box2.add(new JSeparator());
    box3.add(new JTextField(""));
    box4.add(new JSeparator());
    box5.add(new JButton("Bouton 4"));
    box5.add(new JButton("Bouton 5"));
    box5.add(new JButton("bouton 6"));
    add(box1);
    add(box2);
    add(box3);
    add(box4);
    add(box5);
  }

  public static void main(String s[]) {
    JFrame frame = new JFrame("Test separator");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setContentPane(new TestMenuSwing3());
    frame.setMinimumSize(new Dimension(250, 200));
    frame.pack();
    frame.setVisible(true);
  }
}
```

Résultat :

15.10. L'affichage d'une image dans une application.

Pour afficher une image dans une fenêtre, il y a plusieurs solutions.

La plus simple consiste à utiliser le composant JLabel qui est capable d'afficher du texte mais aussi une image

Exemple :

```
package com.jmdoudoux.test;
```

```
import java.awt.BorderLayout;
import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class MonApp  extends JFrame {

  private static final long serialVersionUID = 1L;

  public MonApp(String titre) {
    super(titre);
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    init();
  }

  private void init()
  {
    JLabel label = new JLabel(new ImageIcon("Duke.gif") );
    this.add(label, BorderLayout.CENTER);
    this.pack();
  }

  public static void main(String[] args) {
    MonApp app = new MonApp("Afficher image");
    app.setVisible(true);
  }
}
```

Dans l'exemple ci dessus, le fichier contenant l'image doit être à la racine des fichiers class : aucun chemin n'est précis donc c'est le chemin relatif par rapport au répertoire d'exécution de l'application. Il est possible de préciser un chemin absolu mais cela limite les possibilités de déploiement de l'application.

C:\MonApp\src>javac com/jmdoudoux/test/MonApp.java

C:\MonApp\src>java com.jmdoudoux.test.MonApp

Il est possible de définir un composant personnalisé qui hérite de la classe JPanel qui va se charger d'afficher l'image.

Historiquement, c'est la classe java.awt.Toolkit qui peut être utilisée pour charger une image.

Exemple :

```
package com.jmdoudoux.test;

import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Image;
import java.awt.MediaTracker;
import java.awt.Panel;
import java.awt.Toolkit;

/**
 * Composant qui affiche une image
 */
public class AfficheImage extends Panel {

  private static final long serialVersionUID = 1L;
  private Image image;
```

```
  public AfficheImage(String filename) {
    image = Toolkit.getDefaultToolkit().getImage("./duke.gif");
    try {
      MediaTracker mt = new MediaTracker(this);
      mt.addImage(image, 0);
      mt.waitForAll();
    } catch (Exception e) {
      e.printStackTrace();
    }
    this.setPreferredSize(new Dimension(image.getWidth(this), image
        .getHeight(this)));
  }

  public void paint(Graphics g) {
    g.drawImage(image, 0, 0, null);
  }
}
```

L'inconvénient d'utiliser la classe Toolkit pour charger une image est que ce chargement se fait de façon asynchrone. Il faut alors utiliser une instance de la classe MediaTracker pour patienter le temps de chargement de l'image et ainsi pouvoir déterminer sa taille pour la reporter sur la taille du composant.

A partir de Java 1.4, il est aussi possible d'utiliser la classe javax.imageio.ImageIO pour simplifier le code qui charge l'image.

Exemple :

```
package com.jmdoudoux.test;

import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Panel;
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import javax.imageio.ImageIO;

/**
 * Composant qui affiche une image
 */
public class AfficheImage extends Panel {

  private static final long serialVersionUID = 1L;
  private BufferedImage image;

  public AfficheImage(String nomFichier) {

    try {
      image = ImageIO.read(new File(nomFichier));
      this.setPreferredSize(new Dimension(image.getWidth(),
        image.getHeight()));
    } catch (IOException ie) {
      ie.printStackTrace();
    }
  }

  public void paint(Graphics g) {
    g.drawImage(image, 0, 0, null);
  }
}
```

Il suffit alors d'utiliser le composant dans la fenêtre

Exemple :

```
package com.jmdoudoux.test;
import java.awt.BorderLayout;
```

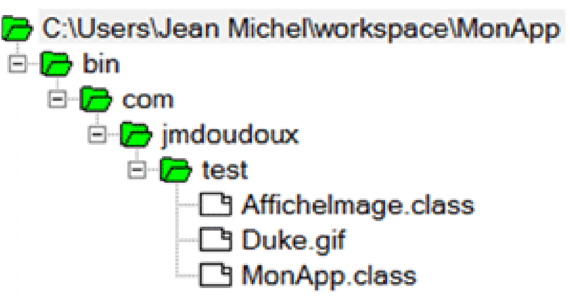
```
import javax.swing.JFrame;

public class MonApp extends JFrame {

  private static final long serialVersionUID = 1L;

  public MonApp(String titre) {
    super(titre);
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    init();
  }

  private void init()
  {
    AfficheImage afficheImage = new AfficheImage("Duke.gif");
    this.setLayout(new BorderLayout());
    this.add(afficheImage, BorderLayout.CENTER);
    this.pack();
  }

  public static void main(String[] args) {
    MonApp app = new MonApp("Afficher image");
    app.setVisible(true);
  }
}
```

Malheureusement, ces deux solutions ne fonctionnent pas si l'application est packagée sous la forme d'une archive qui contient l'image car l'API java.io n'est pas capable de lire une ressource dans l'archive jar. Il faut utiliser le classloader pour charger l'image sous la forme d'une ressource. L'avantage de cette solution c'est qu'elle fonctionne que l'application soit packagée ou non.

Exemple :

```
package com.jmdoudoux.test;

import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Image;
import java.awt.MediaTracker;
import java.awt.Panel;
import java.awt.Toolkit;

/**
 * Composant qui affiche une image
 */
public class AfficheImage extends Panel {

  private static final long serialVersionUID = 1L;
  private Image image;

  public AfficheImage(String filename) {
    java.net.URL url = this.getClass().getResource("Duke.gif");
    image = Toolkit.getDefaultToolkit().getImage(url);
    try {
      MediaTracker mt = new MediaTracker(this);
      mt.addImage(image, 0);
      mt.waitForAll();
    } catch (Exception e) {
      e.printStackTrace();
    }
    this.setPreferredSize(new Dimension(image.getWidth(this), image
        .getHeight(this)));
  }

  public void paint(Graphics g) {
    g.drawImage(image, 0, 0, null);
  }
}
```

Le fichier contenant l'image doit être accessible par le classloader dans le classpath, par exemple :



La suite de ce chapitre sera développée dans une version future de ce document

16. Le développement d'interfaces graphiques avec SWT

Ce chapitre contient plusieurs sections :

- La présentation de SWT
- Un exemple très simple
- La classe SWT
- L'objet Display
- L'objet Shell
- Les composants
- Les conteneurs
- La gestion des erreurs
- Le positionnement des contrôles
- La gestion des événements
- Les boîtes de dialogue

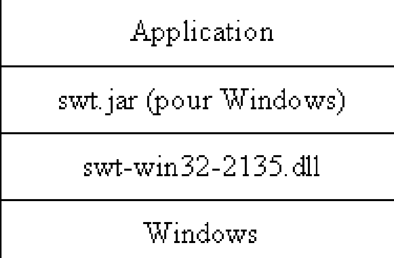
16.1. La présentation de SWT

La première API pour développer des interfaces graphiques portables d'un système à un autre en Java est AWT. Cette API repose sur les composants graphiques du système sous jacent ce qui lui assure de bonne performance. Malheureusement, ces composants sont limités dans leur fonctionnalité car ils représentent le petit dénominateur commun aux communs des différents systèmes concernés.

Pour palier à ce problème, Sun a proposé une nouvelle API, Swing. Cette Api est presque exclusivement écrite en Java, ce qui assure sa portabilité. Swing possède aussi d'autres points forts, tel que des fonctionnalités avancées, la possibilité d'étendre les composants, une adaptation du rendu de composants, etc ... Swing est une API mature, éprouvée et parfaitement connue. Malheureusement, ces deux gros défauts sont sa consommation en ressource machine et la lenteur d'exécution des applications qui l'utilise.

SWT propose une approche intermédiaire : utiliser autant que possible les composants du système et implémenter les autres composants en Java. SWT est écrit en Java et utilise la technologie JNI pour appeler les composants natifs. SWT utilise autant que possible les composants natifs du système lorsque ceux sont présentés, sinon ils sont réécrits en pur Java. Les données de chaque composant sont aussi stockées autant que possible dans le composant natif, limitant ainsi les données stockées dans les objets Java correspondant.

Ainsi, une partie de SWT est livrée sous la forme d'une bibliothèque dépendante du système d'exploitation et d'un fichier .jar lui aussi dépendant du système. Toutes les fonctionnalités de SWT ne sont implémentées que sur les systèmes ou elles sont supportées (exemple, l'utilisation des ActiveX n'est possible que sur le portage de SWT sur les systèmes Windows).



Les trois avantages de SWT sont donc la rapidité d'exécution, des ressources machines moins importantes lors de l'exécution et un rendu parfait des composants graphiques selon le système utilisé puisqu'il utilise des composants natifs. Cette dernière remarque est particulièrement vraie pour des environnements graphiques dont l'apparence est modifiable.

Malgré cette dépendance vis à vis du système graphique de l'environnement d'exécution, l'API de SWT reste la même quelque soit la plate-forme utilisée.

En plus de dépendre du système utilisé lors de l'exécution, SWT possède un autre petit inconvénient. N'utilisant pas de purs objets java, il n'est pas possible de compter sur le ramasse miette pour libérer la mémoire des composants créés manuellement. Pour libérer cette mémoire, il est nécessaire d'utiliser la méthode dispose() pour les composants instanciés lorsque ceux ci ne sont plus utiles.

Pour faciliter ces traitements, l'appel de la méthode dispose() d'un composant entraîne automatiquement l'appel de la méthode dispose() des composants qui lui sont rattachés. Il faut toutefois rester vigilant lors de l'utilisation de certains objets qui ne sont pas des contrôles tels que les objets de type Font ou Color, qu'il convient de libérer explicitement sous peine de fuites de mémoire.

Les règles à observer pour la libération des ressources sont :

- toujours appeler la méthode dispose() de tout objet non rattaché directement à un autre objet qui n'est plus utilisé.
- ne jamais appeler la méthode dispose() d'objets qui n'ont pas été explicitement instanciés dans le code
- l'appel de la méthode dispose() d'un composant entraîne automatiquement l'appel de la méthode dispose() des composants qui lui sont rattachés

Attention, l'utilisation d'un objet dont la méthode dispose() a été appelée induira un résultat imprévisible.

Ainsi SWT repose la problématique concernant la dualité entre la portabilité (Write Once Run Anywhere) et les performances.

SWT repose sur trois concepts classiques dans le développement d'une interface graphique :

- Les composants ou contrôles (widgets)
- Un système de mise en page et de présentation des composants
- Un modèle des gestions des événements

La structure d'une application SWT est la suivante :

- la création d'un objet de type Display qui assure le dialogue avec le système sous jacent
- la création d'un objet de type Shell qui est la fenêtre de l'application
- la création des composants et leur ajout dans le Shell
- l'enregistrement des listeners pour le traitement des événements
- l'exécution de la boucle de gestion des événements jusqu'à la fin de l'application
- la libération des ressources de l'objet Display

La version de SWT utilisée dans ce chapitre est la 2.1.

SWT est regroupé dans plusieurs packages :

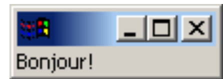
Package	Rôle
org.eclipse.swt	Package de base qui contient la définition de constantes et d'exceptions

org.eclipse.swt.accessibility	
org.eclipse.swt.custom	Contient des composants particuliers
org.eclipse.swt.dnd	Contient les éléments pour le support du « cliqué / glissé »
org.eclipse.swt.events	Contient les éléments pour la gestion des événements
org.eclipse.swt.graphics	Contient les éléments pour l'utilisation des éléments graphiques (couleur, polices, curseur, contexte graphique, ...)
org.eclipse.swt.layout	Contient les éléments pour la gestion de la présentation
org.eclipse.swt.ole.win32	Contient les éléments pour le support d'OLE 32 sous Windows
org.eclipse.swt.printing	Contient les éléments pour le support des impressions
org.eclipse.swt.program	
org.eclipse.swt.widgets	Contient les différents composants

16.2. Un exemple très simple

Exemple :

```
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.*;

public class TestSWT1 {

  public static void main(String[] args) {
    Display display = new Display();
    Shell shell = new Shell(display);

    Label label = new Label(shell, SWT.CENTER);
    label.setText("Bonjour!");
    label.pack();

    shell.pack();
    shell.open();

    while (!shell.isDisposed())
      if (!display.readAndDispatch())
        display.sleep();

    display.dispose();
    label.dispose();
  }
}
```



Pour exécuter cet exemple sous Windows, il faut que le fichier swt.jar correspondant à la plate-forme Windows soit inclus dans le classpath et que l'application puisse accéder à la bibliothèque swt-win32-2135.dll.

16.3. La classe SWT

Cette classe définit un certain nombre de constante concernant les styles. Les styles sont des comportements ou des

caractéristiques définissant l'apparence du composant. Ces styles sont directement fournis dans le constructeur d'une classe encapsulant un composant.

16.4. L'objet Display

Toute application SWT doit obligatoirement instancier un objet de type Display. Cet objet assure le dialogue entre l'application et le système graphique du système d'exploitation utilisé.

Exemple :

```
Display display = new Display();
```

La méthode la plus importante de la classe Display est la méthode readAndDispatch() qui lit les événements dans la pile du système graphique natif pour les diffuser à l'application. Elle renvoie true si il y a encore des traitements a effectuer sinon elle renvoie false.

La méthode sleep() permet de mettre en attente le thread d'écoute de événements jusqu'à l'arrivée d'un nouvel événement.

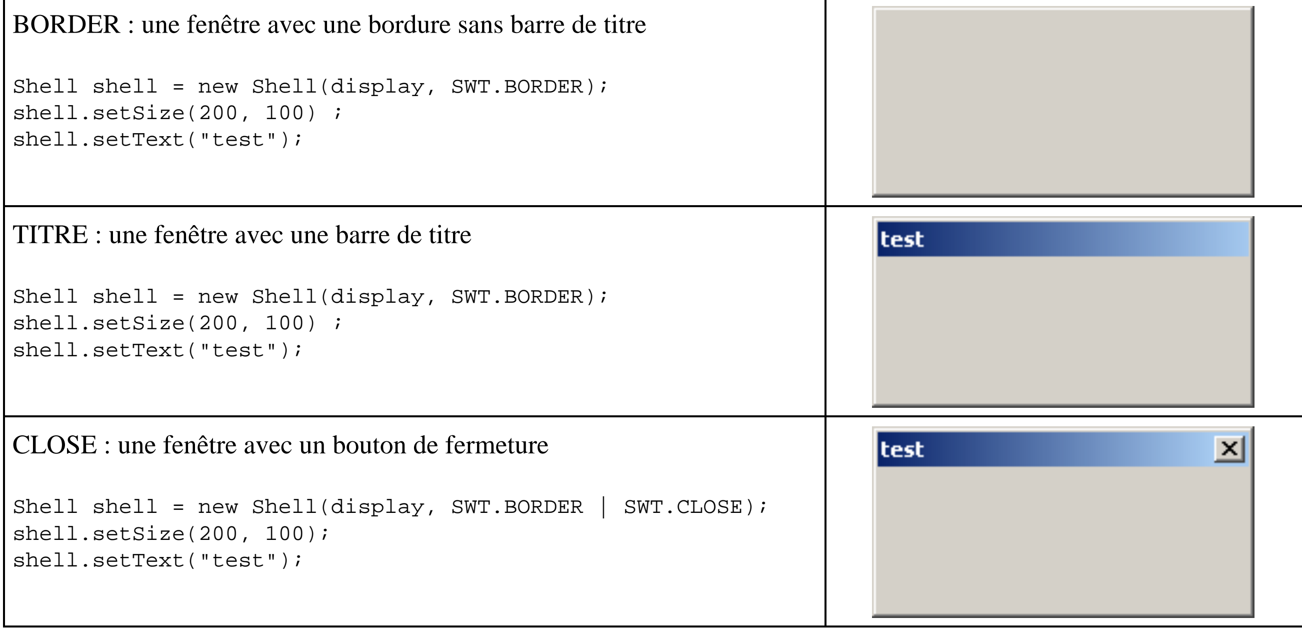
Il est absolument nécessaire lors de la fin de l'application de libérer les ressources allouées par l'objet de type Display en appelant sa méthode dispose().

16.5. L'objet Shell

L'objet Shell représente une fenêtre gérée par le système graphique du système d'exploitation utilisé.

Un objet de type Shell peut être associé à un objet de type Display pour obtenir une fenêtre principale ou être associé à un autre objet de type Shell pour obtenir une fenêtre secondaire.

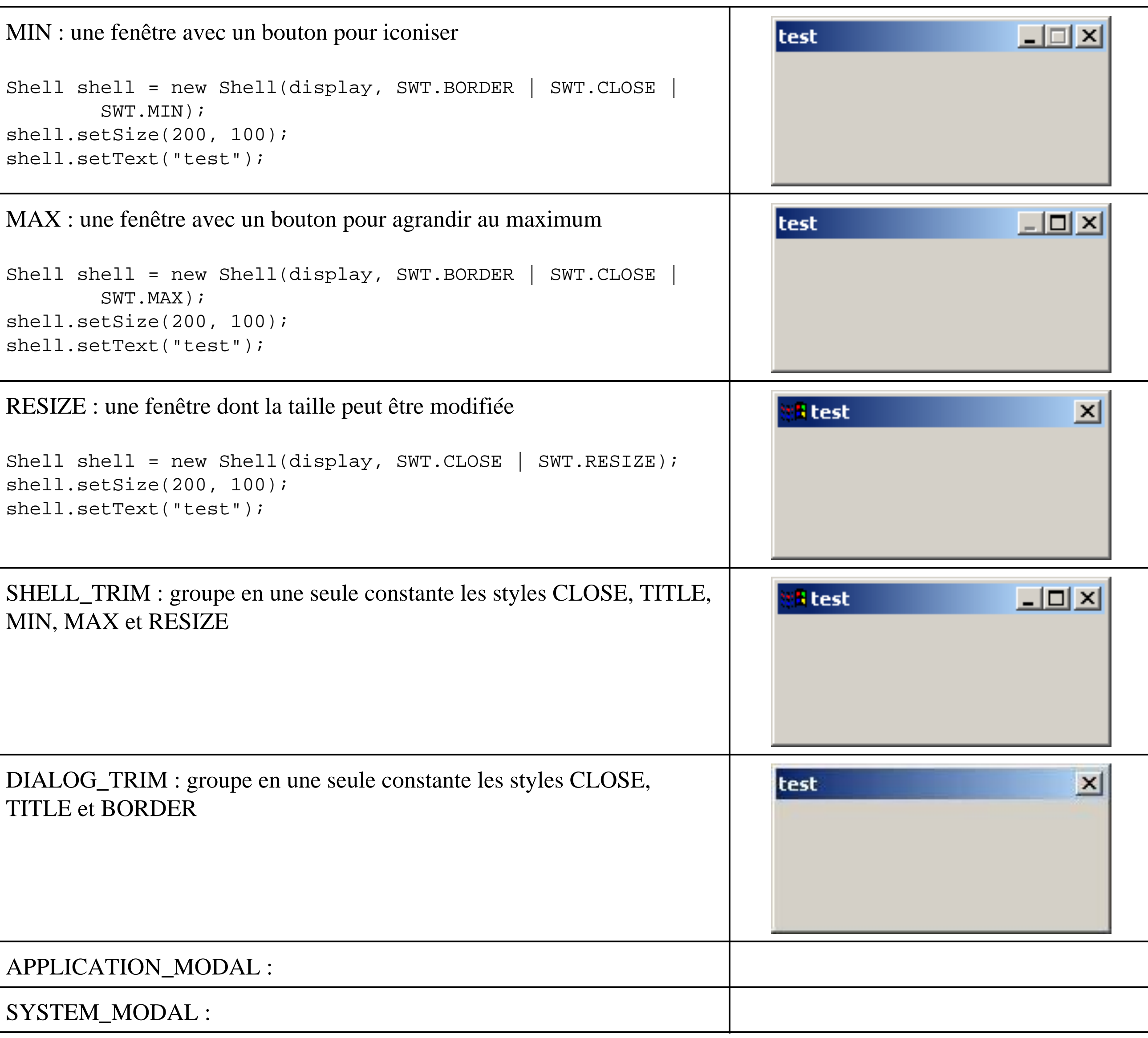
La classe Shell peut utiliser plusieurs styles : BORDER,H_SCROLL, V_SCROLL, CLOSE, MIN, MAX, RESIZE, TITLE, SHELL_TRIM, DIALOG_TRIM



BORDER : une fenêtre avec une bordure sans barre de titre
 	
`Shell shell = new Shell(display, SWT.BORDER);`
`shell.setSize(200, 100) ;`
`shell.setText("test");`	

TITRE : une fenêtre avec une barre de titre
 	
`Shell shell = new Shell(display, SWT.BORDER);`
`shell.setSize(200, 100) ;`
`shell.setText("test");`	

CLOSE : une fenêtre avec un bouton de fermeture
 	
`Shell shell = new Shell(display, SWT.BORDER | SWT.CLOSE);`
`shell.setSize(200, 100);`
`shell.setText("test");`	



Style	Aperçu

MIN : une fenêtre avec un bouton pour iconiser
 	
`Shell shell = new Shell(display, SWT.BORDER | SWT.CLOSE |`
`        SWT.MIN);`
`shell.setSize(200, 100);`
`shell.setText("test");`
	test

MAX : une fenêtre avec un bouton pour agrandir au maximum
 	
`Shell shell = new Shell(display, SWT.BORDER | SWT.CLOSE |`
`        SWT.MAX);`
`shell.setSize(200, 100);`
`shell.setText("test");`
	test

RESIZE : une fenêtre dont la taille peut être modifiée
 	
`Shell shell = new Shell(display, SWT.CLOSE | SWT.RESIZE);`
`shell.setSize(200, 100);`
`shell.setText("test");`
	test

SHELL_TRIM : groupe en une seule constante les styles CLOSE, TITLE, MIN, MAX et RESIZE	test
DIALOG_TRIM : groupe en une seule constante les styles CLOSE, TITLE et BORDER	test
APPLICATION_MODAL :	
SYSTEM_MODAL :	

La méthode setSize() permet de préciser la taille de la fenêtre.

La méthode setTexte() permet de préciser le titre de la fenêtre.

Exemple : centrer la fenêtre sur l'écran

```
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.graphics.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.*;

public class TestSWT1 {

  public static void centrerSurEcran(Display display, Shell shell) {
    Rectangle rect = display.getClientArea();
    Point size = shell.getSize();
    int x = (rect.width - size.x) / 2;
    int y = (rect.height - size.y) / 2;
    shell.setLocation(new Point(x, y));
  }

  public static void main(String[] args) {
    Display display = new Display();
    Shell shell = new Shell(display);
    shell.setSize(340, 100);
    centrerSurEcran(display, shell);
```

```
    shell.open();

    while (!shell.isDisposed())
      if (!display.readAndDispatch())
        display.sleep();

    display.dispose();
  }
}
```

16.6. Les composants

Les composants peuvent être regroupés en deux grandes familles :

- les contrôles qui sont des composants graphiques. Ils héritent tous de la classe abstraite Control
- les conteneurs qui permettent de grouper des contrôles. Ils héritent tous de la classe abstraite Composite

Une application SWT est une hiérarchie de composants dont la racine est un objet de type Shell.

Certaines caractéristiques comme l'apparence ou le comportement d'un contrôle doivent être fournies au moment de leur création par le système graphique. Ainsi, chaque composant SWT possède une propriété nommée style fournie en paramètre du constructeur.

Plusieurs styles peuvent être combinés avec l'opérateur | . Cependant certains styles sont incompatibles entre eux pour certains composants.

16.6.1. La classe Control

La classe Control définit trois styles : BORDER, LEFT_TO_RIGHT et RIGHT_TO_LEFT

Le seul constructeur de la classe Control nécessite aussi de préciser le composant père sous la forme d'un objet de type Composite. L'association avec le composant père est obligatoire pour tous les composants lors de leur création.

La classe Control possède plusieurs méthodes pour enregistrer des listeners pour certains événements. Ces événements sont : FocusIn, FocusOut, Help, KeyDown, KeyUp, MouseDoubleClick, MouseDown, MouseEnter, MouseExit, MouseHover, MouseUp, MouseMove, Move, Paint, Resize.

Elle possède aussi plusieurs méthodes dont les principales sont :

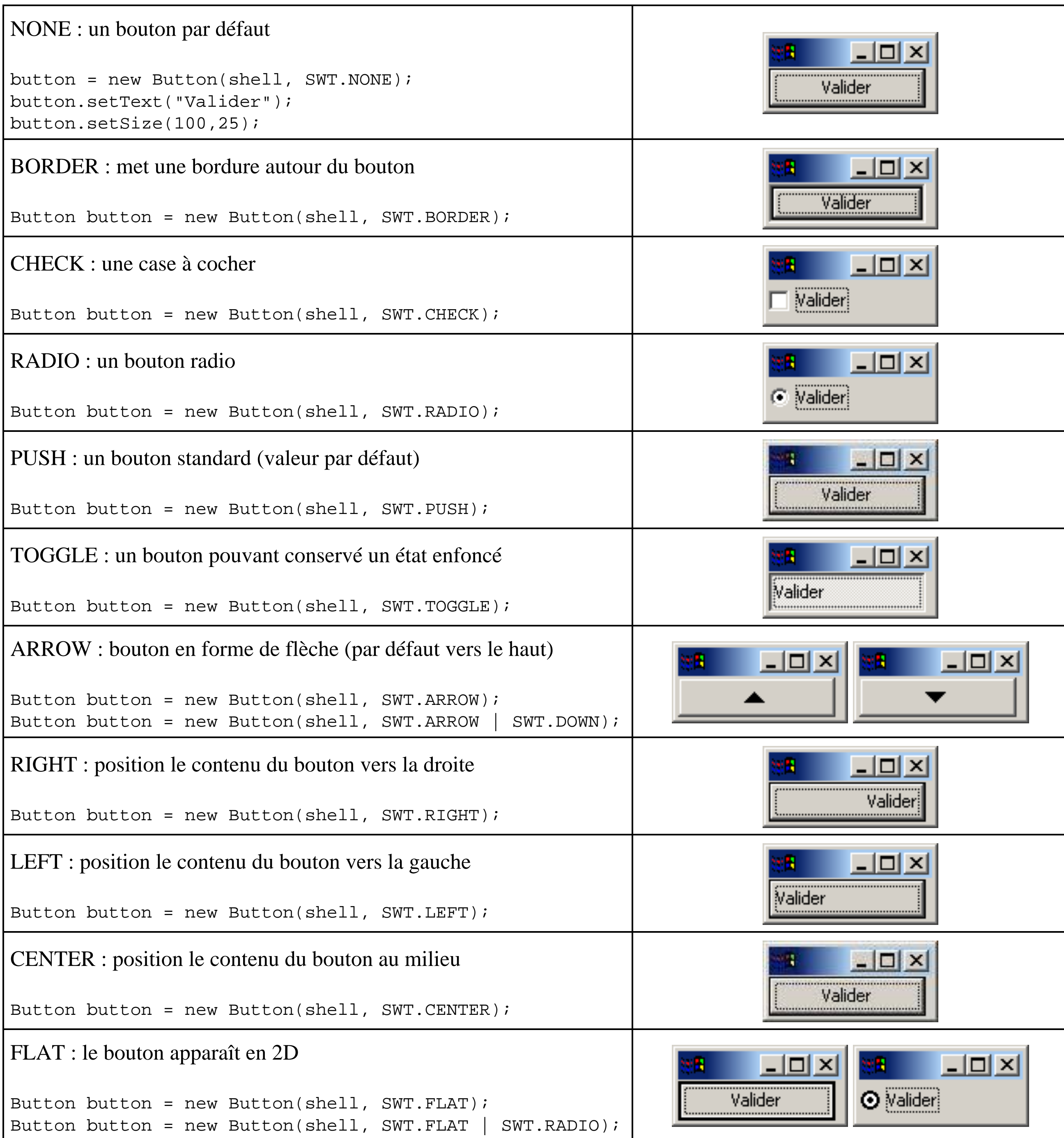
Nom	Rôle
boolean forceFocus()	Force le focus au composant pour lui permettre de recevoir les événements clavier
Display getDisplay()	Renvoie l'objet Display associé au composant
Shell getShell()	Renvoie l'objet Shell associé au composant
void pack()	Recalcule la taille préférée du composant
void SetEnabled()	Permet de rendre actif le composant
void SetFocus()	Donne le focus au composant pour lui permettre de recevoir les événements clavier
void setSize()	Permet de modifier la taille du composant
void setVisible()	Permet de rendre visible ou non le composant

16.6.2. Les contrôles de base

16.6.2.1. La classe Button

La classe Button représente un bouton cliquable.

La classe Button définit plusieurs styles : BORDER, CHECK, PUSH, RADIO, TOGGLE, FLAT, LEFT, RIGHT, CENTER, ARROW (avec UP, DOWN)



NONE : un bouton par défaut
 	
`button = new Button(shell, SWT.NONE);`
`button.setText("Valider");`
`button.setSize(100,25);`
	Valider

BORDER : met une bordure autour du bouton `Button button = new Button(shell, SWT.BORDER);`	Valider
CHECK : une case à cocher `Button button = new Button(shell, SWT.CHECK);`	Valider
RADIO : un bouton radio `Button button = new Button(shell, SWT.RADIO);`	Valider
PUSH : un bouton standard (valeur par défaut) `Button button = new Button(shell, SWT.PUSH);`	Valider
TOGGLE : un bouton pouvant conservé un état enfoncé `Button button = new Button(shell, SWT.TOGGLE);`	Valider

ARROW : bouton en forme de flèche (par défaut vers le haut)
 	
`Button button = new Button(shell, SWT.ARROW);`
`Button button = new Button(shell, SWT.ARROW | SWT.DOWN);`	

RIGHT : position le contenu du bouton vers la droite `Button button = new Button(shell, SWT.RIGHT);`	Valider
LEFT : position le contenu du bouton vers la gauche `Button button = new Button(shell, SWT.LEFT);`	Valider
CENTER : position le contenu du bouton au milieu `Button button = new Button(shell, SWT.CENTER);`	Valider

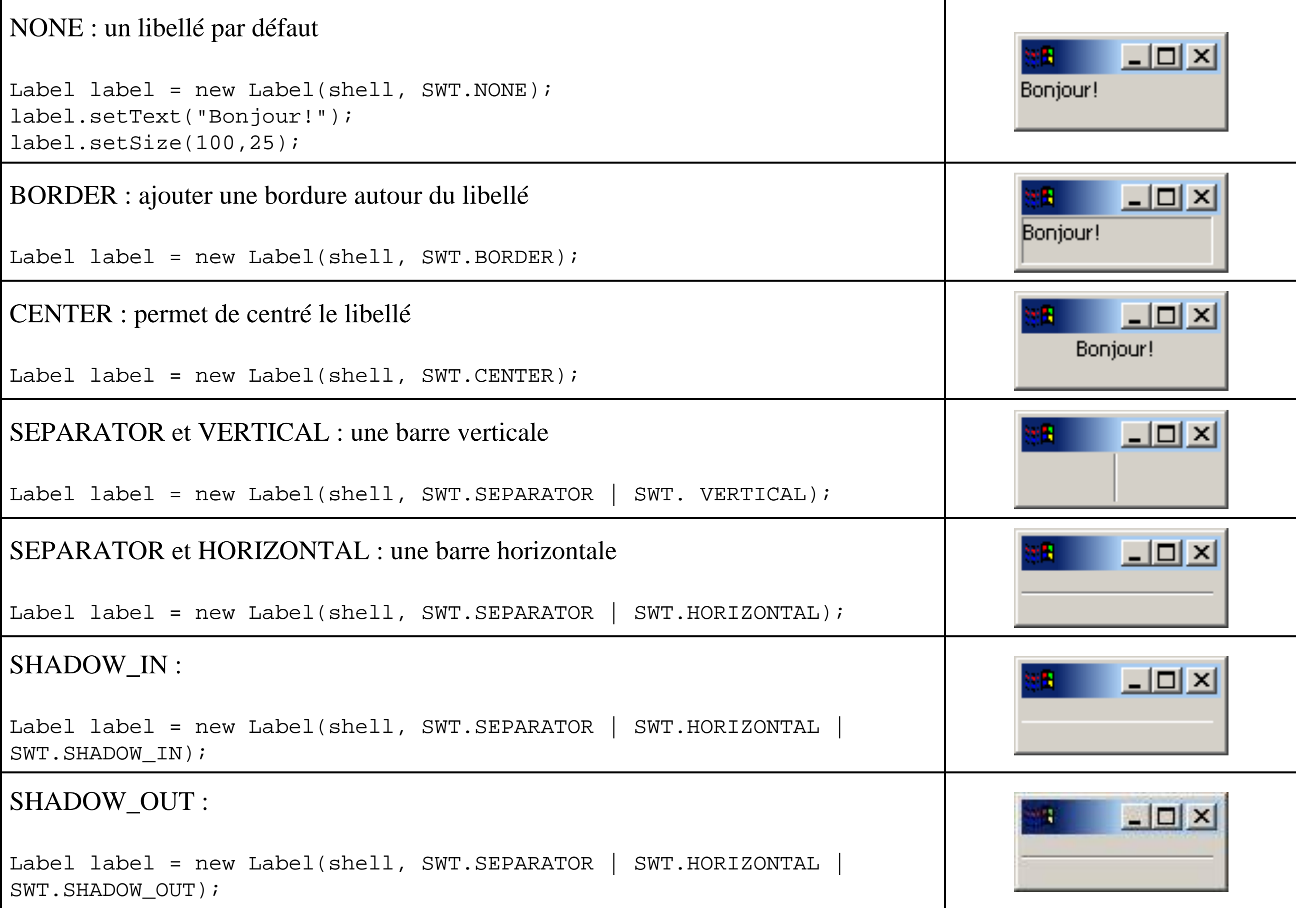
FLAT : le bouton apparaît en 2D
 	
`Button button = new Button(shell, SWT.FLAT);`
`Button button = new Button(shell, SWT.FLAT | SWT.RADIO);`
	Valider Valider

16.6.2.2. La classe Label

Ce contrôle permet d'afficher un libellé ou une image

La classe Label possède plusieurs styles : BORDER, CENTER, LEFT, RIGHT, WRAP, SEPARATOR (avec

HORIZONTAL, SHADOW_IN, SHADOW_OUT, SHADOW_NONE, VERTICAL)



Style	Aperçu

NONE : un libellé par défaut
`Label label = new Label(shell, SWT.NONE);`
`label.setText("Bonjour!");`
`label.setSize(100,25);`
	Bonjour!

BORDER : ajouter une bordure autour du libellé `Label label = new Label(shell, SWT.BORDER);`	Bonjour!
CENTER : permet de centré le libellé `Label label = new Label(shell, SWT.CENTER);`	Bonjour!
SEPARATOR et VERTICAL : une barre verticale `Label label = new Label(shell, SWT.SEPARATOR | SWT. VERTICAL);`	
SEPARATOR et HORIZONTAL : une barre horizontale `Label label = new Label(shell, SWT.SEPARATOR | SWT.HORIZONTAL);`	

SHADOW_IN :
`Label label = new Label(shell, SWT.SEPARATOR | SWT.HORIZONTAL |`
`SWT.SHADOW_IN);`	

SHADOW_OUT :
`Label label = new Label(shell, SWT.SEPARATOR | SWT.HORIZONTAL |`
`SWT.SHADOW_OUT);`	

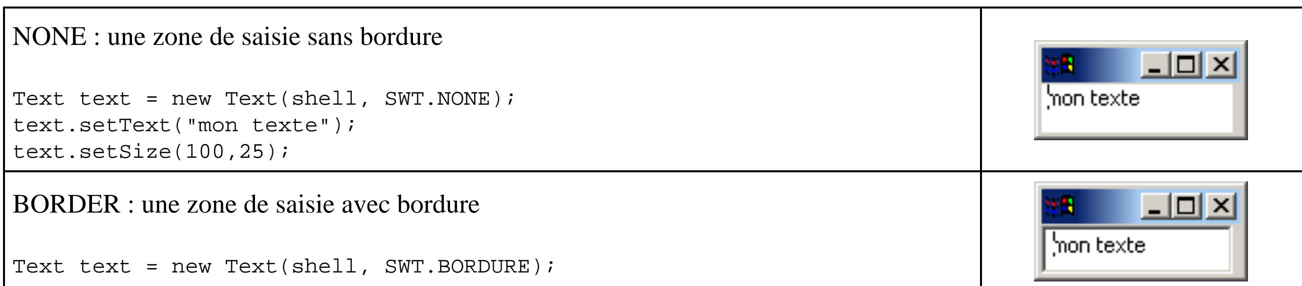
Cette classe possède plusieurs méthodes dont les principales sont :

Nom	Rôle
void setAlignment(int)	Permet de préciser l'alignement des données du contrôle
void setImage(Image)	Permet de préciser une image affichée par le contrôle
void setText(string)	Permet de préciser le texte du contrôle

16.6.2.3. La classe Text

Ce contrôle est une zone de saisie de texte.

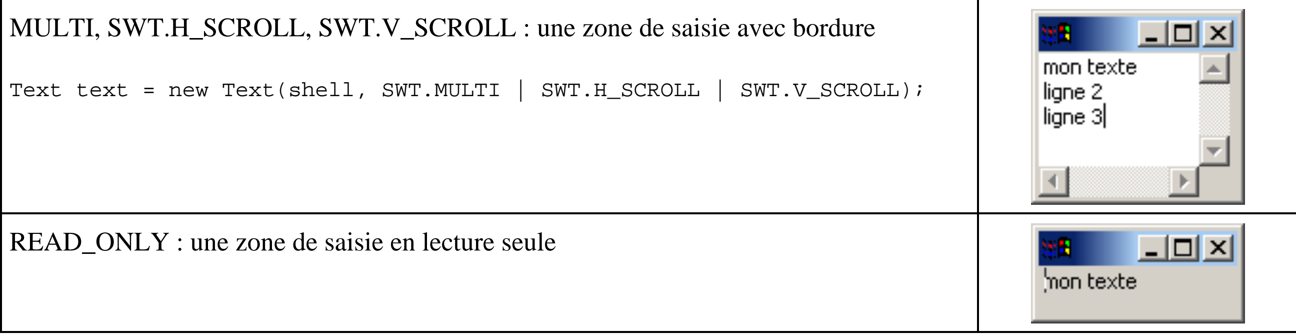
La classe Text possède plusieurs styles : BORDER, SINGLE, READ_ONLY, LEFT, CENTER, RIGHT, WRAP, MULTI (avec H_SCROLL, V_SCROLL)



Style	Aperçu

NONE : une zone de saisie sans bordure
`Text text = new Text(shell, SWT.NONE);`
`text.setText("mon texte");`
`text.setSize(100,25);`
	mon texte

BORDER : une zone de saisie avec bordure `Text text = new Text(shell, SWT.BORDURE);`	mon texte



MULTI, SWT.H_SCROLL, SWT.V_SCROLL : une zone de saisie avec bordure

```
Text text = new Text(shell, SWT.MULTI | SWT.H_SCROLL | SWT.V_SCROLL);
```


READ_ONLY : une zone de saisie en lecture seule

Cette classe possède plusieurs méthodes dont les principales sont :

Nom	Rôle
void setEchoChar(char)	Caractère affiché lors de la frappe d'une touche
void setTextLimit(int)	Permet de préciser le nombre maximum de caractères saisissable
void setText(string)	Permet de préciser le contenu de la zone de texte
void setEditable(boolean)	Permet de rendre le contrôle éditable ou non

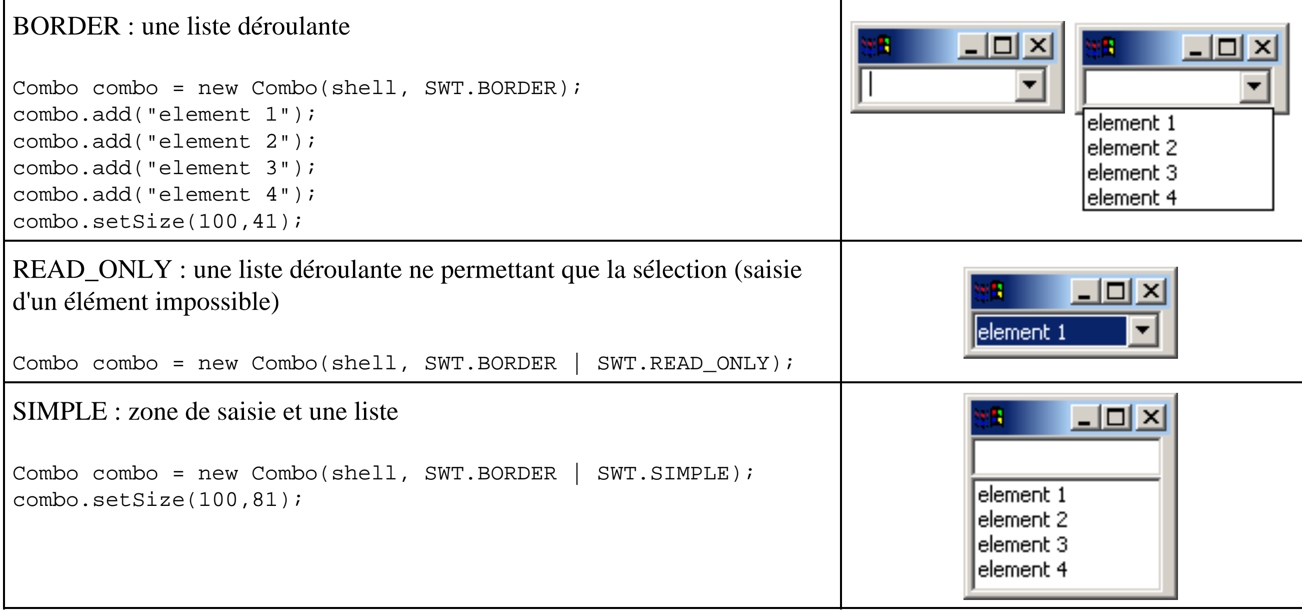
16.6.3. Les contrôles de type liste

SWT permet de créer de type liste et liste déroulante.

16.6.3.1. La classe Combo

Ce contrôle est une liste déroulante dans laquelle l'utilisateur peut sélectionner une valeur dans une liste d'éléments prédéfinis ou saisir un élément.

La classe Combo définit trois styles : BORDER, DROP_DOWN, READ_ONLY, SIMPLE



BORDER : une liste déroulante

```
Combo combo = new Combo(shell, SWT.BORDER);
combo.add("element 1");
combo.add("element 2");
combo.add("element 3");
combo.add("element 4");
combo.setSize(100,41);
```


READ_ONLY : une liste déroulante ne permettant que la sélection (saisie d'un élément impossible)

```
Combo combo = new Combo(shell, SWT.BORDER | SWT.READ_ONLY);
```

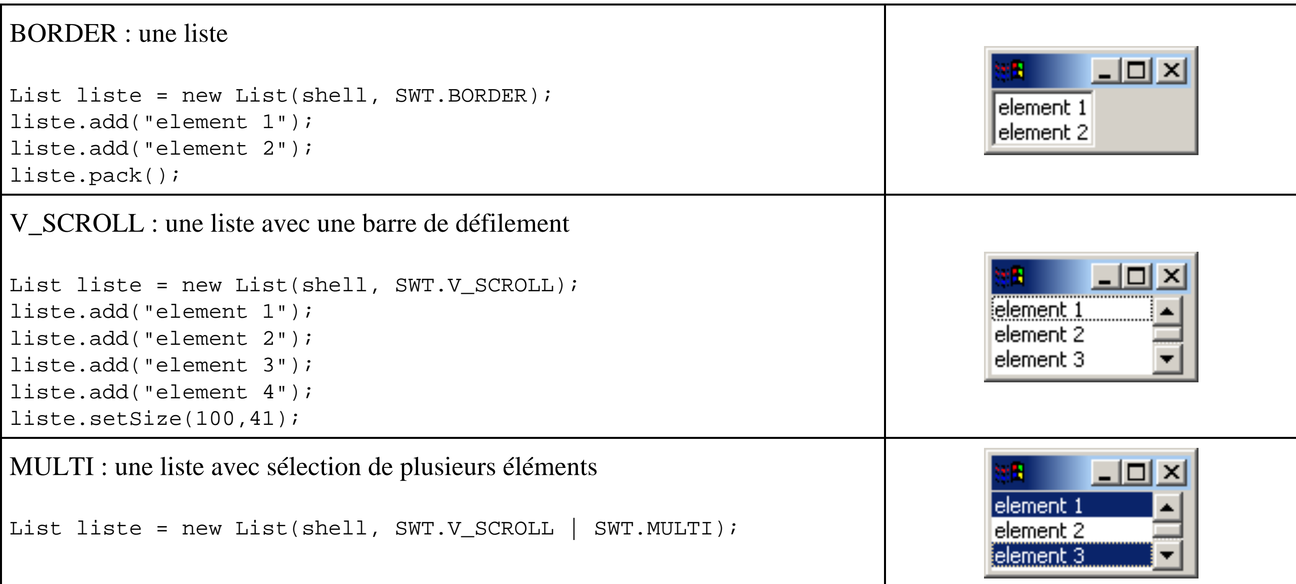

SIMPLE : zone de saisie et une liste

```
Combo combo = new Combo(shell, SWT.BORDER | SWT.SIMPLE);
combo.setSize(100,81);
```

16.6.3.2. La classe List

Ce contrôle est une liste qui permet de sélectionner un ou plusieurs éléments.

La classe List possède plusieurs styles : BORDER, H_SCROLL, V_SCROLL, SINGLE, MULTI



BORDER : une liste

```
List liste = new List(shell, SWT.BORDER);
liste.add("element 1");
liste.add("element 2");
liste.pack();
```

V_SCROLL : une liste avec une barre de défilement

```
List liste = new List(shell, SWT.V_SCROLL);
liste.add("element 1");
liste.add("element 2");
liste.add("element 3");
liste.add("element 4");
liste.setSize(100,41);
```

MULTI : une liste avec sélection de plusieurs éléments

```
List liste = new List(shell, SWT.V_SCROLL | SWT.MULTI);
```

La méthode add() permet d'ajouter un élément à la liste sous la forme d'un chaîne de caractères.

La méthode setItems() permet de fournir les éléments de la liste sous la forme d'un tableau de chaîne de caractères.

Exemple :

```
List liste = new List(shell, SWT.V_SCROLL | SWT.MULTI);
liste.setItems(new String[] {"element 1", "element 2", "element 3", "element 4"});
liste.setSize(100,41);
```

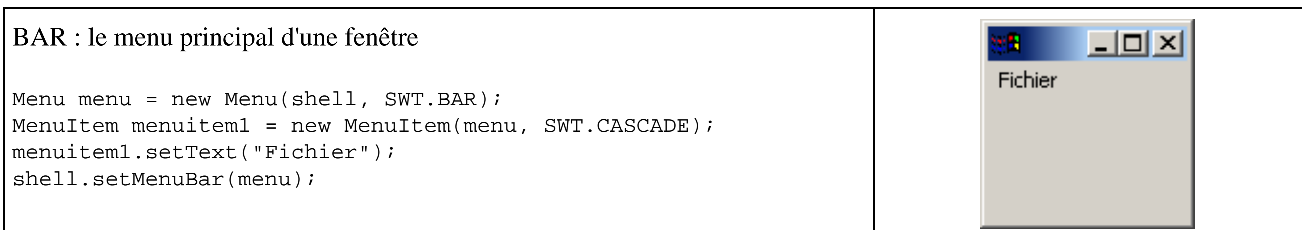
16.6.4. Les contrôles pour les menus

SWT permet la création de menus principaux et de menus déroulants. La création de ces menus met en oeuvre deux classes : Menu, MenuItem

16.6.4.1. La classe Menu

Ce contrôle est un élément du menu qui va contenir des options

La classe Menu possède plusieurs styles : BAR, DROP_DOWN, NO_RADIO_GROUP, POP_UP



BAR : le menu principal d'une fenêtre

```
Menu menu = new Menu(shell, SWT.BAR);
MenuItem menuitem1 = new MenuItem(menu, SWT.CASCADE);
menuitem1.setText("Fichier");
shell.setMenuBar(menu);
```

POP_UP : un menu contextuel

```
Menu menu = new Menu(shell, SWT.POP_UP);
MenuItem menuitem1 = new MenuItem(menu, SWT.CASCADE);
menuitem1.setText("Fichier");
MenuItem menuitem2 = new MenuItem(menu, SWT.CASCADE);
menuitem2.setText("Aide");
shell.setMenu(menu);
```

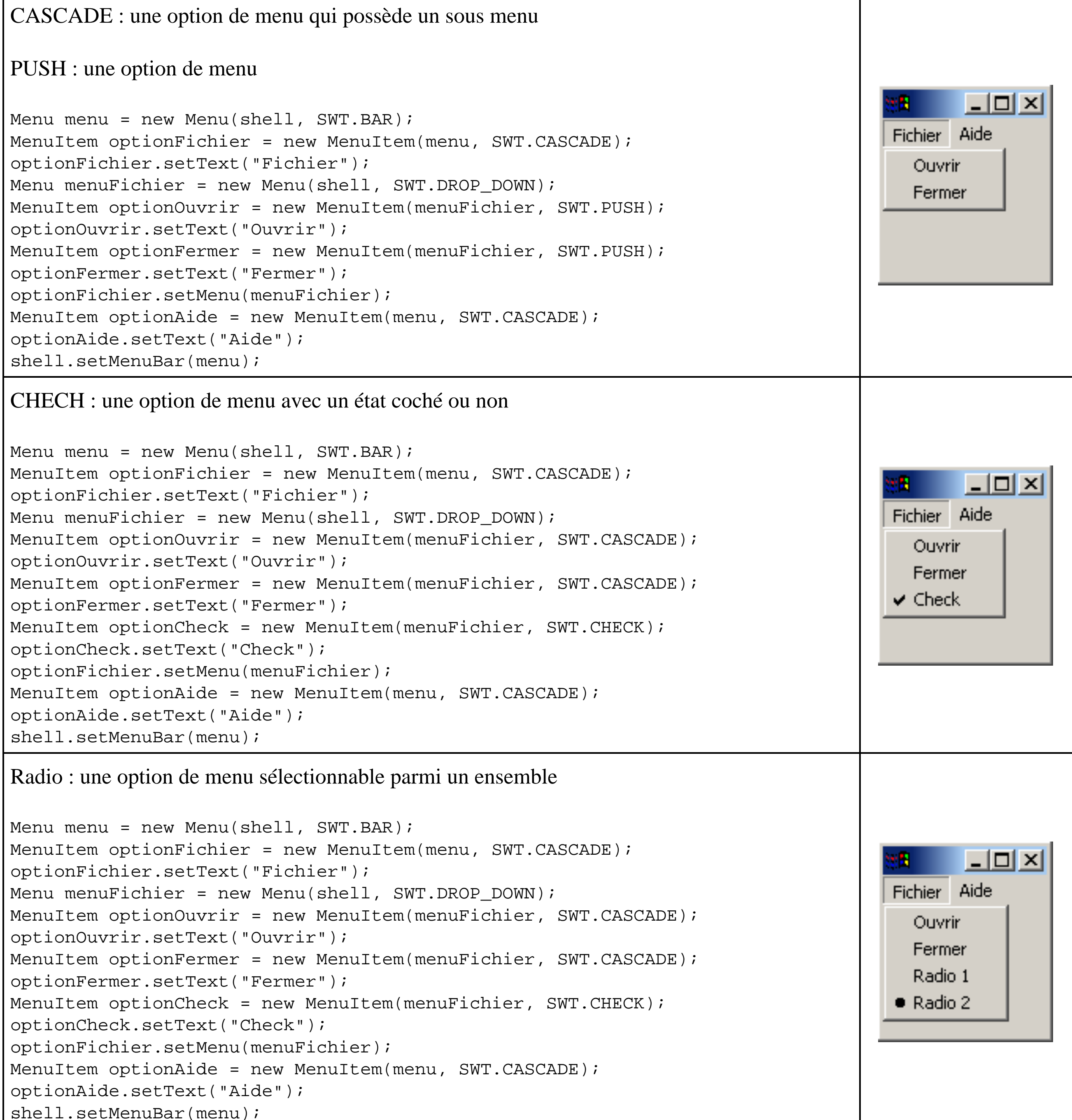
DROP_DOWN : un sous menu

NO_RADIO_GROUP :

16.6.4.2. La classe MenuItem

Ce contrôle est une option d'un menu.

La classe MenuItem possède styles : CHECK, CASCADE, PUSH, RADIO, SEPARATOR



CASCADE : une option de menu qui possède un sous menu

PUSH : une option de menu

```
Menu menu = new Menu(shell, SWT.BAR);
MenuItem optionFichier = new MenuItem(menu, SWT.CASCADE);
optionFichier.setText("Fichier");
Menu menuFichier = new Menu(shell, SWT.DROP_DOWN);
MenuItem optionOuvrir = new MenuItem(menuFichier, SWT.PUSH);
optionOuvrir.setText("Ouvrir");
MenuItem optionFermer = new MenuItem(menuFichier, SWT.PUSH);
optionFermer.setText("Fermer");
optionFichier.setMenu(menuFichier);
MenuItem optionAide = new MenuItem(menu, SWT.CASCADE);
optionAide.setText("Aide");
shell.setMenuBar(menu);
```

CHECH : une option de menu avec un état coché ou non

```
Menu menu = new Menu(shell, SWT.BAR);
MenuItem optionFichier = new MenuItem(menu, SWT.CASCADE);
optionFichier.setText("Fichier");
Menu menuFichier = new Menu(shell, SWT.DROP_DOWN);
MenuItem optionOuvrir = new MenuItem(menuFichier, SWT.CASCADE);
optionOuvrir.setText("Ouvrir");
MenuItem optionFermer = new MenuItem(menuFichier, SWT.CASCADE);
optionFermer.setText("Fermer");
MenuItem optionCheck = new MenuItem(menuFichier, SWT.CHECK);
optionCheck.setText("Check");
optionFichier.setMenu(menuFichier);
MenuItem optionAide = new MenuItem(menu, SWT.CASCADE);
optionAide.setText("Aide");
shell.setMenuBar(menu);
```

Radio : une option de menu sélectionnable parmi un ensemble

```
Menu menu = new Menu(shell, SWT.BAR);
MenuItem optionFichier = new MenuItem(menu, SWT.CASCADE);
optionFichier.setText("Fichier");
Menu menuFichier = new Menu(shell, SWT.DROP_DOWN);
MenuItem optionOuvrir = new MenuItem(menuFichier, SWT.CASCADE);
optionOuvrir.setText("Ouvrir");
MenuItem optionFermer = new MenuItem(menuFichier, SWT.CASCADE);
optionFermer.setText("Fermer");
MenuItem optionCheck = new MenuItem(menuFichier, SWT.CHECK);
optionCheck.setText("Check");
optionFichier.setMenu(menuFichier);
MenuItem optionAide = new MenuItem(menu, SWT.CASCADE);
optionAide.setText("Aide");
shell.setMenuBar(menu);
```

SEPARATOR : une option de menu sous la forme d'un séparateur

```
Menu menu = new Menu(shell, SWT.BAR);
MenuItem optionFichier = new MenuItem(menu, SWT.CASCADE);
optionFichier.setText("Fichier");
Menu menuFichier = new Menu(shell, SWT.DROP_DOWN);
MenuItem optionOuvrir = new MenuItem(menuFichier, SWT.PUSH);
optionOuvrir.setText("Ouvrir");
MenuItem optionSeparator = new MenuItem(menuFichier, SWT.SEPARATOR);
MenuItem optionFermer = new MenuItem(menuFichier, SWT.PUSH);
optionFermer.setText("Fermer");
optionFichier.setMenu(menuFichier);
MenuItem optionAide = new MenuItem(menu, SWT.CASCADE);
optionAide.setText("Aide");
shell.setMenuBar(menu);
```

La méthode setText() permet de préciser le libellé de l'option de menu.

La méthode setAccelerator() permet de préciser un raccourci clavier.

```
Menu menuFichier = new Menu(shell, SWT.DROP_DOWN);
MenuItem optionOuvrir = new MenuItem(menuFichier, SWT.PUSH);
optionOuvrir.setText("&Ouvrir\tCtrl+O");
optionOuvrir.setAccelerator(SWT.CTRL+'O');
```

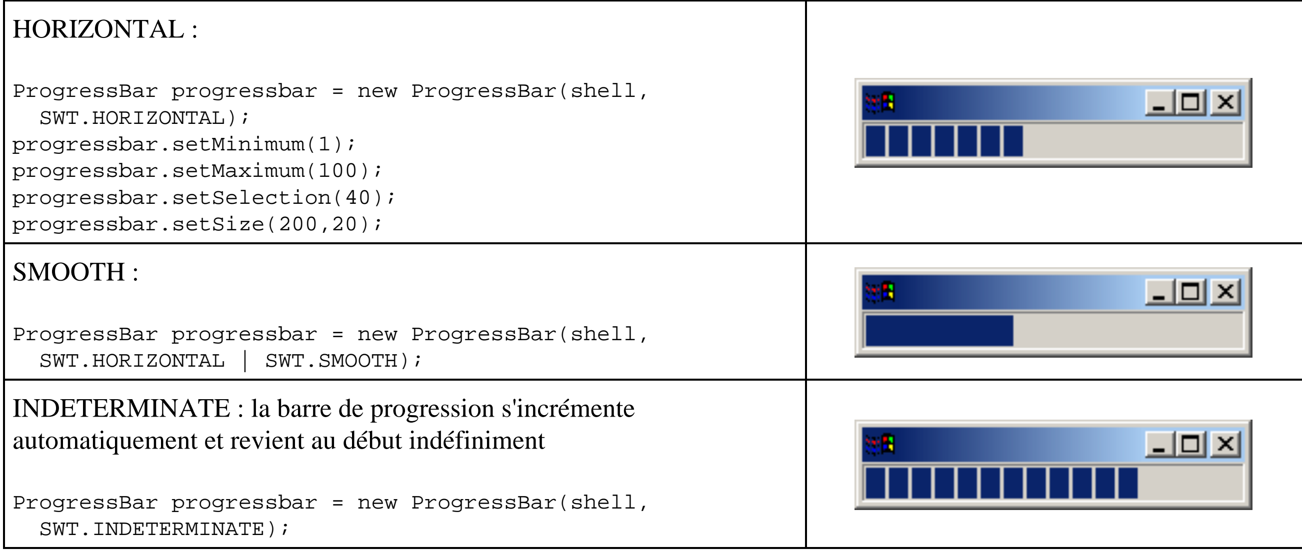
16.6.5. Les contrôles de sélection ou d'affichage d'une valeur

SWT propose un contrôle pour l'affichage d'une barre de progression et deux contrôles pour la sélection d'une valeur numérique dans une plage de valeur.

16.6.5.1. La classe ProgressBar

Ce contrôle est une barre de progression.

La classe ProgressBar possède plusieurs styles : BORDER, INDETERMINATE, SMOOTH, HORIZONTAL, VERTICAL



HORIZONTAL :

```
ProgressBar progressbar = new ProgressBar(shell,
  SWT.HORIZONTAL);
progressbar.setMinimum(1);
progressbar.setMaximum(100);
progressbar.setSelection(40);
progressbar.setSize(200,20);
```

SMOOTH :

```
ProgressBar progressbar = new ProgressBar(shell,
  SWT.HORIZONTAL | SWT.SMOOTH);
```

INDETERMINATE : la barre de progression s'incrémente automatiquement et revient au début indéfiniment

```
ProgressBar progressbar = new ProgressBar(shell,
  SWT.INDETERMINATE);
```

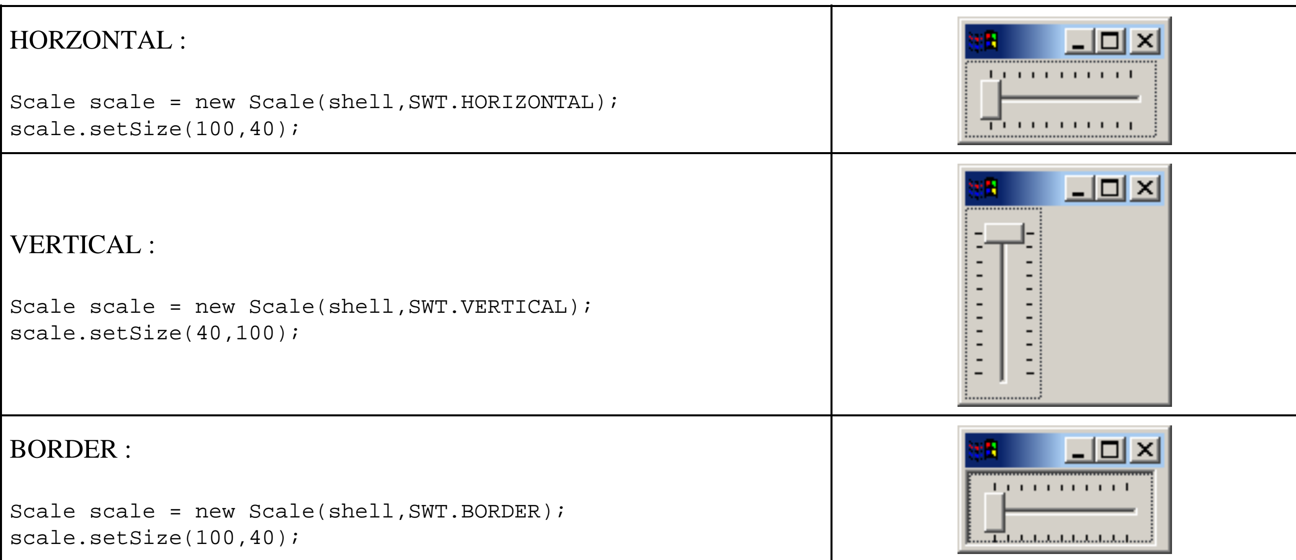
Les méthodes setMinimum() et setMaximum() permettent respectivement de préciser la valeur minimale et la valeur maximale du contrôle.

La méthode setSelection() permet de positionner la valeur courante de l'indicateur.

16.6.5.2. La classe Scale

Ce contrôle permet de sélectionner une valeur dans une plage valeur numérique.

La classe Scale possède trois styles : BORDER, HORIZONTAL, VERTICAL



HORZONTAL :
`Scale scale = new Scale(shell,SWT.HORIZONTAL);`
`scale.setSize(100,40);`	

VERTICAL :
`Scale scale = new Scale(shell,SWT.VERTICAL);`
`scale.setSize(40,100);`	

BORDER :
`Scale scale = new Scale(shell,SWT.BORDER);`
`scale.setSize(100,40);`	

Les méthodes setMinimum() et setMaximum() permettent respectivement de préciser la valeur minimale et la valeur maximale du contrôle.

La méthode setSelection() permet de positionner le curseur dans la plage de valeur à la valeur fournie en paramètre.

La méthode setPageIncrement() permet de préciser la valeur fourni en paramètre d'incrémentation d'une page

Exemple :

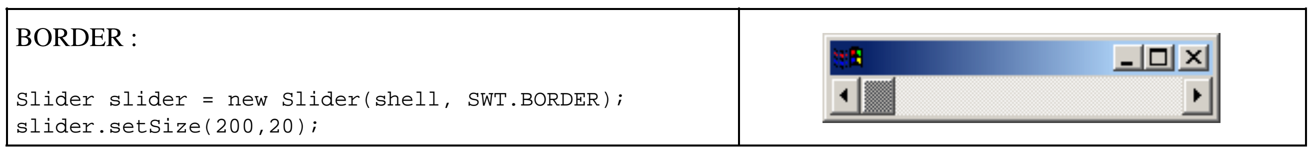
Scale scale = new Scale(shell,SWT.HORIZONTAL);

```
scale.setSize(100,40);
scale.setMinimum(1);
scale.setMaximum(100);
scale.setSelection(30);
scale.setPageIncrement(10);
```

16.6.5.3. La classe Slider

Ce contrôle permet de sélectionner une valeur dans une plage valeur numérique.

La classe Slider possède trois styles : BORDER, HORIZONTAL, VERTICAL



BORDER :
`Slider slider = new Slider(shell, SWT.BORDER);`
`slider.setSize(200,20);`	

VERTICAL :
 	
`Slider slider = new Slider(shell, SWT.VERTICAL);`
`slider.setSize(20,200);`	

Les méthodes setMinimum() et setMaximum() permettent respectivement de préciser la valeur minimale et la valeur maximale du contrôle.

La méthode setSelection() permet de positionner le curseur dans la plage de valeur à la valeur fournie en paramètre.

La méthode setPageIncrement() permet de préciser la valeur fourni en paramètre d'incrémentation d'une page

La méthode setThumb() permet de préciser la taille du curseur.

Exemple :

Slider slider = new Slider(shell,SWT.HORIZONTAL);
 	
`slider.setMinimum(1);`
`slider.setMaximum(110);`
`slider.setSelection(30);`
`slider.setThumb(10);`
`slider.setSize(100,20);`	

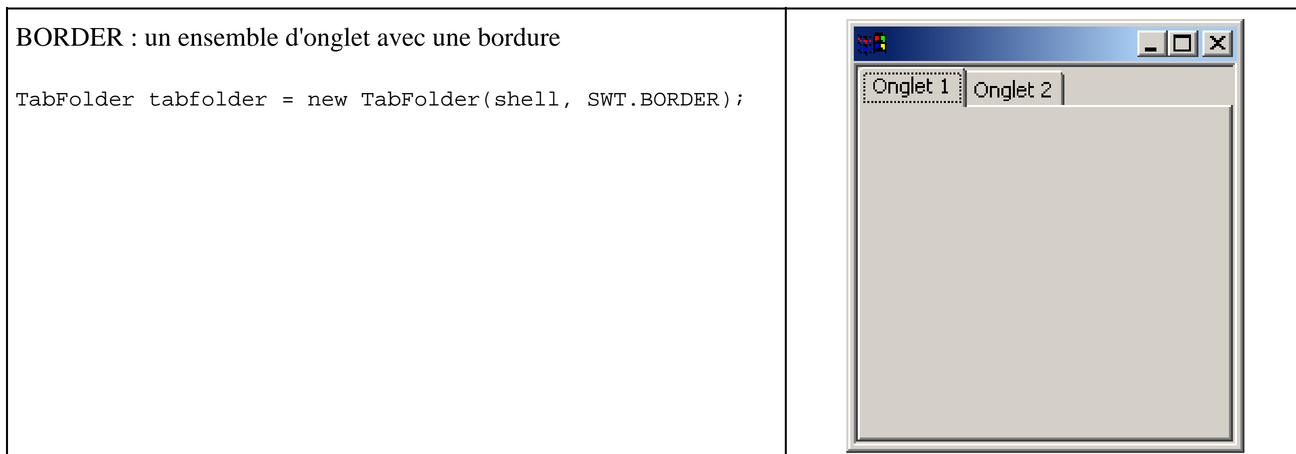
16.6.6. Les contrôles de type « onglets »

SWT propose la création de composants de type onglets mettant en oeuvre deux classes : TabFolder et TabItem

16.6.6.1. La classe TabFolder

Ce contrôle est un ensemble d'onglets.

NONE :
 	
`TabFolder tabfolder = new TabFolder(shell, SWT.NONE);`
`tabfolder.setSize(200,200);`
`TabItem onglet1 = new TabItem(tabfolder, SWT.NONE);`
`onglet1.setText("Onglet 1");`
`TabItem onglet2 = new TabItem(tabfolder, SWT.NONE);`
`onglet2.setText("Onglet 2");`	



BORDER : un ensemble d'onglet avec une bordure

```
TabFolder tabfolder = new TabFolder(shell, SWT.BORDER);
```

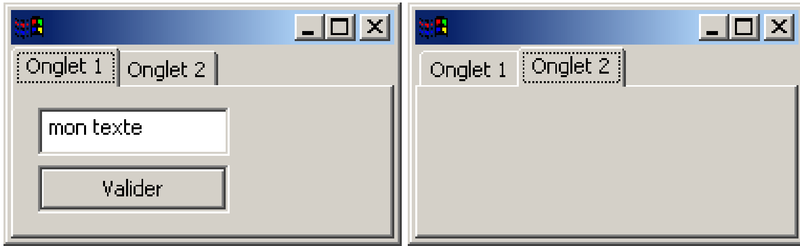

16.6.6.2. La classe TabItem

Ce contrôle est un onglet d'un ensemble d'onglets

Il n'est possible d'insérer qu'un seul contrôle dans un onglet grâce la commande setControl(). Rl est ainsi pratique de regrouper les différents dans un contrôle de type Composite.

Exemple :

```
TabFolder tabfolder = new TabFolder(shell, SWT.NONE);
tabfolder.setSize(200,100);
TabItem onglet1 = new TabItem(tabfolder, SWT.NONE);
onglet1.setText("Onglet 1");
TabItem onglet2 = new TabItem(tabfolder, SWT.NONE);
onglet2.setText("Onglet 2");

Composite pageOnglet1 = new Composite(tabfolder, SWT.NONE);
Text text1 = new Text(pageOnglet1, SWT.BORDER);
text1.setText("mon texte");
text1.setBounds(10,10,100,25);
Button button1 = new Button(pageOnglet1, SWT.BORDER);
button1.setText("Valider");
button1.setBounds(10,40,100,25);
onglet1.setControl(pageOnglet1);
```



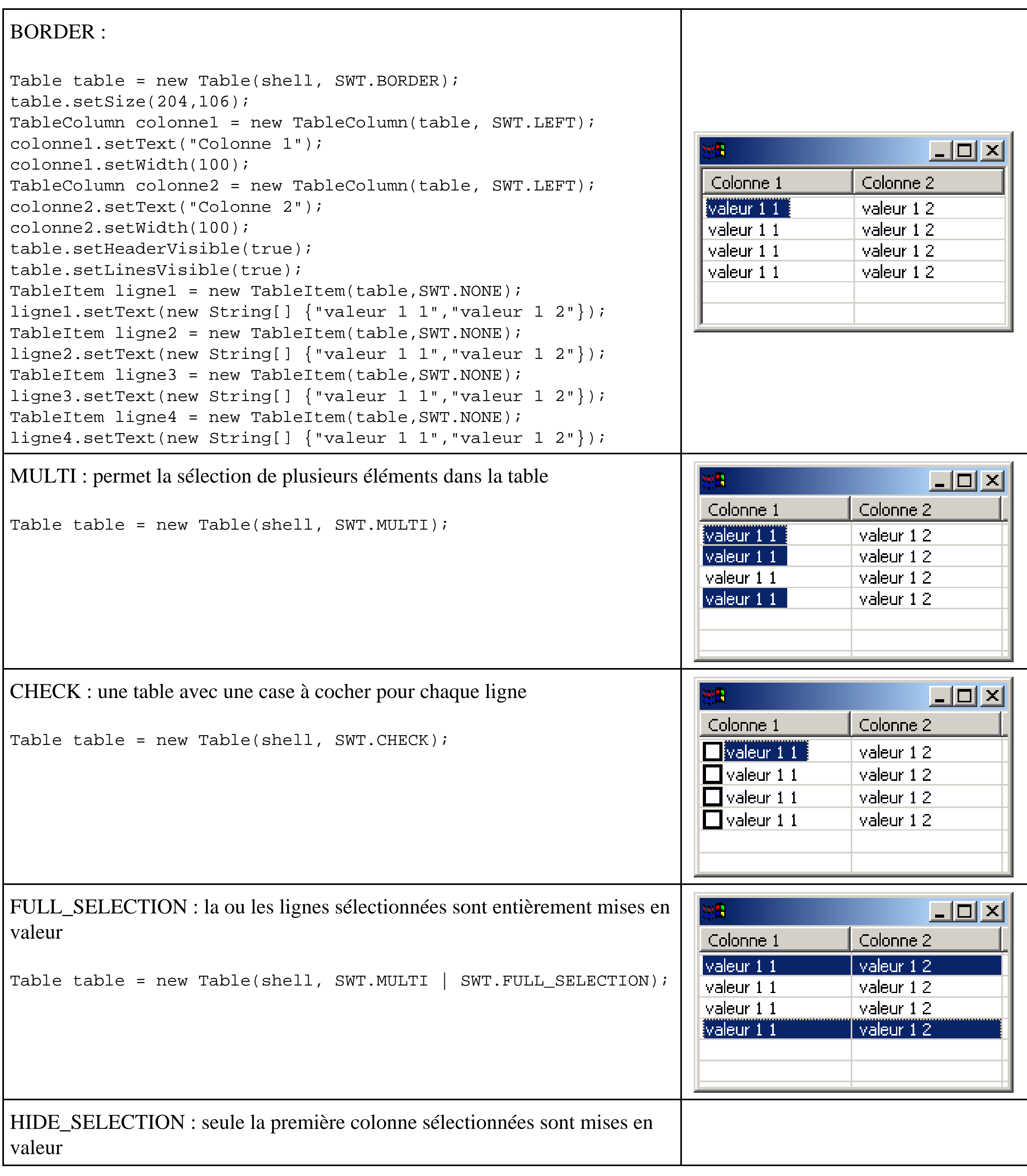
16.6.7. Les contrôles de type « tableau »

SWT permet la création d'un contrôle de type tableau pour afficher et sélectionner des données en mettant en oeuvre trois classes : Table, TableColumn et TableItem.

16.6.7.1. La classe Table

Ce contrôle permet d'afficher et de sélectionner des éléments sous la forme d'un tableau.

La classe Table possède plusieurs styles : BORDER, H_SCROLL, V_SCROLL, SINGLE, MULTI, CHECK, FULL_SELECTION, HIDE_SELECTION



BORDER :

```
Table table = new Table(shell, SWT.BORDER);
table.setSize(204,106);
TableColumn colonne1 = new TableColumn(table, SWT.LEFT);
colonne1.setText("Colonne 1");
colonne1.setWidth(100);
TableColumn colonne2 = new TableColumn(table, SWT.LEFT);
colonne2.setText("Colonne 2");
colonne2.setWidth(100);
table.setHeaderVisible(true);
table.setLinesVisible(true);
TableItem ligne1 = new TableItem(table,SWT.NONE);
ligne1.setText(new String[] {"valeur 1 1","valeur 1 2"});
TableItem ligne2 = new TableItem(table,SWT.NONE);
ligne2.setText(new String[] {"valeur 1 1","valeur 1 2"});
TableItem ligne3 = new TableItem(table,SWT.NONE);
ligne3.setText(new String[] {"valeur 1 1","valeur 1 2"});
TableItem ligne4 = new TableItem(table,SWT.NONE);
ligne4.setText(new String[] {"valeur 1 1","valeur 1 2"});
```

MULTI : permet la sélection de plusieurs éléments dans la table

```
Table table = new Table(shell, SWT.MULTI);
```

CHECK : une table avec une case à cocher pour chaque ligne

```
Table table = new Table(shell, SWT.CHECK);
```

FULL_SELECTION : la ou les lignes sélectionnées sont entièrement mises en valeur

```
Table table = new Table(shell, SWT.MULTI | SWT.FULL_SELECTION);
```

HIDE_SELECTION : seule la première colonne sélectionnées sont mises en valeur

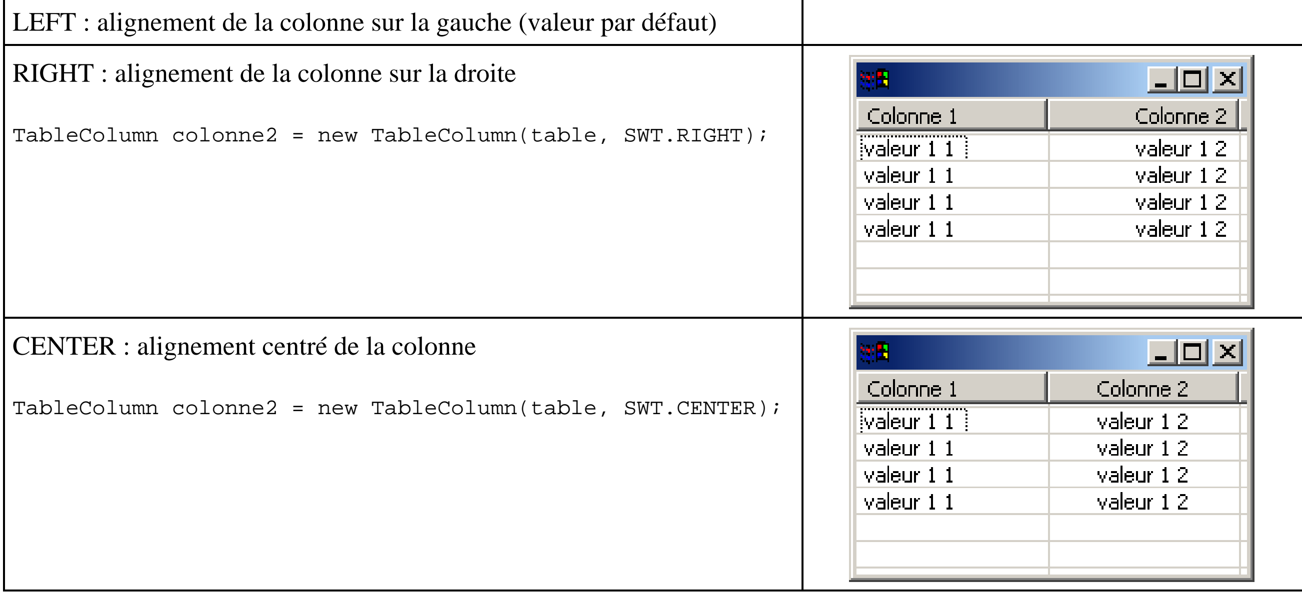
La méthode setHeaderVisible() permet de préciser si l'en tête de la table doit être affichée ou non : par défaut sa valeur est non affichée (false).

La méthode setLinesVisible() permet de préciser si lignes de la table doivent être affichées ou non : par défaut sa valeur est non affiché (false).

16.6.7.2. La classe TableColumn

Ce contrôle est une colonne d'un contrôle Table

La classe TableColumn possède trois styles : LEFT, RIGHT, CENTER



LEFT : alignement de la colonne sur la gauche (valeur par défaut)	
RIGHT : alignement de la colonne sur la droite `TableColumn colonne2 = new TableColumn(table, SWT.RIGHT);`	Colonne 1 / Colonne 2 valeur 1 1 / valeur 1 2 valeur 1 1 / valeur 1 2 valeur 1 1 / valeur 1 2 valeur 1 1 / valeur 1 2
CENTER : alignement centré de la colonne `TableColumn colonne2 = new TableColumn(table, SWT.CENTER);`	Colonne 1 / Colonne 2 valeur 1 1 / valeur 1 2 valeur 1 1 / valeur 1 2 valeur 1 1 / valeur 1 2 valeur 1 1 / valeur 1 2

Bizarement, seul le style LEFT semble pouvoir s'appliquer à la première colonne de la table.

La méthode setWidth() permet de préciser la largeur de la colonne

La méthode setText() permet de préciser le libellé d'en tête de la colonne

La méthode setResizable() permet de préciser si la colonne peut être redimensionnée ou non.

16.6.7.3. La classe TableItem

Ce contrôle est une ligne d'un contrôle Table

La classe TableItem ne possède aucun style.

Il existe plusieurs surcharges de la méthode setText() pour fournir à chaque ligne les données de ces colonnes.

Une surcharge de cette méthode permet de fournir les données sous la forme d'un tableau de chaînes de caractères.

Exemple :

```
ligne1.setText(new String[] {"valeur 1 1","valeur 1 2"});
```

Une autre surcharge de cette méthode permet de préciser le numéro de la colonne et le texte. La première colonne possède le numéro 0.

Exemple : modifier la valeur de la première cellule de la ligne

```
ligne4.setText(0 ,"valeur 2 2");
```

La méthode setCheck() permet de cocher ou non la case associée à la ligne si la table possède le style CHECK.

16.6.8. Les contrôles de type « arbre »

SWT permet la création d'un composant de type arbre en mettant en oeuvre les classes Tree et TreeItem.

16.6.8.1. La classe Tree

Ce contrôle affiche et permet de sélectionner des données sous la forme d'une arborescence

La classe Tree possède plusieurs styles : BORDER, H_SCROLL, V_SCROLL, SINGLE, MULTI, CHECK

SINGLE : un arbre avec sélection unique

```
Tree tree = new Tree(shell, SWT.SINGLE);
TreeItem tree_1 = new TreeItem(tree, SWT.NONE);
tree_1.setText("element 1");
TreeItem tree_2 = new TreeItem(tree, SWT.NONE);
tree_2.setText("element 2");
TreeItem tree_2_1 = new TreeItem(tree_2, SWT.NONE);
tree_2_1.setText("element 2 1");
TreeItem tree_2_2 = new TreeItem(tree_2, SWT.NONE);
tree_2_2.setText("element 2 2");
TreeItem tree_3 = new TreeItem(tree, SWT.NONE);
tree_3.setText("element 3");
tree.setSize(100, 100);
```

BORDER : arbre avec une bordure

```
Tree tree = new Tree(shell, SWT.SINGLE | SWT.BORDER);
```

H_SCROLL et V_SCROLL : arbre avec si nécessaire une barre de défilement respectivement horizontal et vertical

```
Tree tree = new Tree(shell, SWT.SINGLE | SWT.BORDER |
  SWT.H_SCROLL | SWT.V_SCROLL);
```

MULTI : un arbre avec sélection multiple possible

```
Tree tree = new Tree(shell, SWT.MULTI | SWT.BORDER);
```

CHECK : un arbre avec une case à cocher devant chaque élément

```
Tree tree = new Tree(shell, SWT.CHECK | SWT.BORDER);
```

16.6.8.2. La classe TreeItem

Ce contrôle est un élément d'une arborescence

Cette classe ne possède pas de style particulier.

Pour ajouter un élément racine à l'arbre, il suffit de passer l'arbre en tant qu'élément conteneur dans le constructeur.

Pour ajouter un élément fils à un élément, il suffit de passer l'élément père en tant qu'élément conteneur dans le constructeur.

Il existe un constructeur qui attend un troisième paramètre permettant de préciser la position de l'élément.

Exemple :

```
Tree tree = new Tree(shell, SWT.SINGLE);
TreeItem tree_1 = new TreeItem(tree, SWT.NONE);
tree_1.setText("element 1");
TreeItem tree_2 = new TreeItem(tree, SWT.NONE);
tree_2.setText("element 2");
TreeItem tree_2_1 = new TreeItem(tree_2, SWT.NONE);
tree_2_1.setText("element 2 1");
TreeItem tree_2_2 = new TreeItem(tree_2, SWT.NONE);
tree_2_2.setText("element 2 2");
TreeItem tree_3 = new TreeItem(tree, SWT.NONE);
tree_3.setText("element 3");
tree.setSize(100, 100);
```

16.6.9. La classe ScrollBar

Ce contrôle est une barre de défilement

La classe ScrollBar possède deux styles : HORIZONTAL, VERTICAL

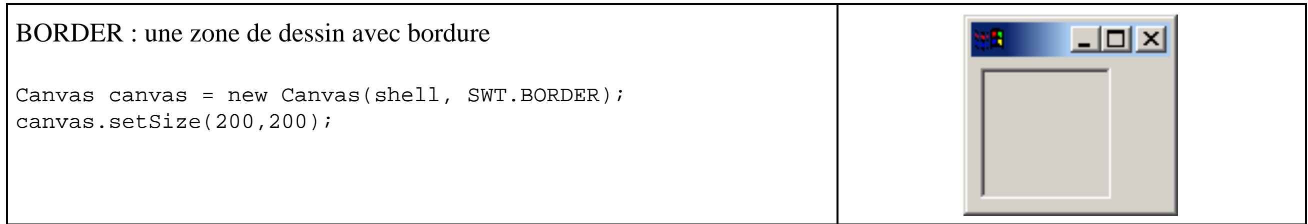
16.6.10. Les contrôles pour le graphisme

SWT permet de dessiner des formes graphiques en mettant en oeuvre la classe GC et la classe Canvas.

16.6.10.1. La classe Canvas

Ce contrôle est utilisé pour dessiner des formes graphiques

La classe Canvas définit plusieurs styles : BORDER, H_SCROLL, V_SCROLL, NO_BACKGROUND, NO_FOCUS, NO_MERGE_PAINTS, NO_REDRAW_RESIZE, NO_RADIO_GROUP



BORDER : une zone de dessin avec bordure

```
Canvas canvas = new Canvas(shell, SWT.BORDER);
canvas.setSize(200,200);
```

16.6.10.2. La classe GC

Cette classe encapsule un contexte graphique dans lequel il va être possible de dessiner des formes graphiques.

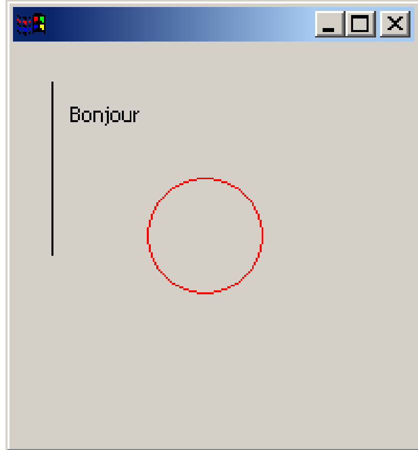
Pour réaliser ces opérations, la classe GC propose de nombreuses méthodes.

Attention : il est important d'appeler la méthode open() de la fenêtre avant de réaliser des opérations de dessin sur le contexte.

Ne pas oublier le libérer les ressources allouées à la classe GC en utilisant la méthode dispose() si l'objet de GC est explicitement instancié dans le code.

Exemple :

```
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.graphics.*;

public class TestSWT21 {

  public static void main(String[] args) {
    final Display display = new Display();
    final Shell shell = new Shell(display);
    shell.setSize(420,420);

    Canvas canvas = new Canvas(shell, SWT.NONE);
    canvas.setSize(200,200);
    canvas.setLocation(10,10);
    shell.pack();
    shell.open();

    GC gc = new GC(canvas);
    gc.drawText("Bonjour",20,20);
    gc.drawLine(10,10,10,100);
    gc.setForeground(display.getSystemColor(SWT.COLOR_RED));
    gc.drawOval(60,60,60,60);
    gc.dispose();

    while (!shell.isDisposed()) {
      if (!display.readAndDispatch())
        display.sleep();
    }
    display.dispose();
  }
}
```



Dans cette exemple, le dessin est réalisé une seule fois au démarrage, il n'est donc pas redessiné si nécessaire (fenêtre partiellement ou complètement masquée, redimensionnement, ...). Pour résoudre ce problème, il faut mettre les opérations de dessin en réponse à un événement de type PaintListener.

16.6.10.3. La classe Color

Cette classe encapsule une couleur définie dans le système graphique.

Elle possède deux constructeurs qui attendent en paramètre l'objet de type Display et soit un objet de type RGB, soit trois entiers représentant les valeurs des couleurs rouge, vert et bleu.

La classe RGB encapsule simplement les trois entiers représentant les valeurs des couleurs rouge, vert et bleu.

Exemple :

```
Color couleur = new Color(display,155,0,0);
Color couleur = new Color(display, new RGB(155,0,0));
```

Remarque : il ne faut pas oublier d'utiliser la méthode dispose() pour libérer les ressources du système allouées à cet objet une fois que celui ci n'est plus utilisé.

Exemple :

```
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.graphics.*;

public class TestSWT22 {

  public static void main(String[] args) {
    final Display display = new Display();
    final Shell shell = new Shell(display);

    shell.setSize(200, 200);
    Color couleur = new Color(display,155,0,0);
    shell.setBackground(couleur);
    shell.open();

   while (!shell.isDisposed()) {
      if (!display.readAndDispatch())
        display.sleep();
    }

    couleur.dispose();
    display.dispose();
  }
}
```

16.6.10.4. La classe Font

Cette classe encapsule une police de caractère définie dans le système graphique.

La classe Font peut utiliser plusieurs styles : NORMAL, BOLD et ITALIC

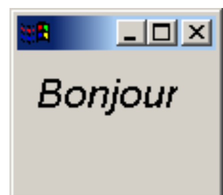
Il existe plusieurs constructeurs dont le plus simple à utiliser nécessite en paramètre l'objet display, le nom de la police (celle ci doit être présente sur le système), la taille et le style.

Remarque : il ne faut pas oublier d'utiliser la méthode dispose() pour libérer les ressources du système allouées à cet objet une fois que celui ci n'est plus utilisé.

Exemple :

```
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.graphics.*;
import org.eclipse.swt.*;

public class TestSWT23 {
```

```
  public static void main(String[] args) {
    final Display display = new Display();
    final Shell shell = new Shell(display);

    Font font = new Font(display, "Arial", 16, SWT.ITALIC);
    Label label = new Label(shell, SWT.NONE);
    label.setFont(font);
    label.setText("Bonjour");
    label.setLocation(10, 10);
    label.pack();
    shell.setSize(100, 100);
    shell.open();

    while (!shell.isDisposed()) {
      if (!display.readAndDispatch())
        display.sleep();
    }

    font.dispose();
    display.dispose();
  }
}
```



16.6.10.5. La classe Image

Cette classe encapsule une image au format BMP, ICO, GIF, JPEG ou PNG.

La classe Image possède plusieurs constructeurs dont le plus simple à utiliser est celui nécessitant en paramètres l'objet Display et une chaîne de caractères contenant le chemin vers le fichier de l'image

Remarque : il ne faut pas oublier d'utiliser la méthode dispose() pour libérer les ressources du système allouées à cet objet une fois que celui ci n'est plus utilisé.

Exemple :

```
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.graphics.*;
import org.eclipse.swt.*;

public class TestSWT24 {

  public static void main(String[] args) {
    final Display display = new Display();
    final Shell shell = new Shell(display);

    Image image = new Image(display, "btn1.bmp");
    Button bouton = new Button(shell, SWT.FLAT);
    bouton.setImage(image);
    bouton.setBounds(10, 10, 50, 50);
    shell.setSize(100, 100);
    shell.open();

    while (!shell.isDisposed()) {
      if (!display.readAndDispatch())
        display.sleep();
    }

    image.dispose();
    display.dispose();
```

```
  }
}
```

Si l'application doit être packagée dans un fichier jar, incluant les images utiles, il faut utiliser la getResourceAsStream() du classloader pour charger l'image.

Exemple :

```
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.graphics.*;
import org.eclipse.swt.*;
import java.io.*;

public class TestSWT25 {

  public static void main(String[] args) {
    final Display display = new Display();
    final Shell shell = new Shell(display);

    InputStream is = TestSWT25.class.getResourceAsStream("btn1.bmp");
    Image image = new Image(display, is);
    Button bouton = new Button(shell, SWT.FLAT);
    bouton.setImage(image);
    bouton.setBounds(10, 10, 50, 50);
    shell.setSize(100, 100);
    shell.open();

    while (!shell.isDisposed()) {
      if (!display.readAndDispatch())
        display.sleep();
    }

    image.dispose();
    display.dispose();
  }
}
```

16.7. Les conteneurs

Ce type de composant permet de contenir d'autres contrôles.

16.7.1. Les conteneurs de base

SWT propose deux contrôles de ce type : Composite et Group.

16.7.1.1. La classe Composite

Ce contrôle est un conteneur pour d'autres contrôles.

Ce contrôle possède les styles particuliers suivants : BORDER, H_SCROLL et V_SCROLL

BORDER : permet la présence d'une bordure autour du composant `Composite composite = new Composite(shell, SWT.NONE);`	
H_SCROLL : permet la présence d'une barre de défilement horizontal `Composite composite = new Composite(shell, SWT.H_SCROLL);`	
V_SCROLL : permet la présence d'une barre de défilement vertical `Composite composite = new Composite(shell, SWT.V_SCROLL);`	

Les contrôles sont ajoutés au contrôle Composite de la même façon que dans un objet de type Shell en précisant simplement que le conteneur est l'objet de type Composite.

La position indiquée pour les contrôles inclus dans le Composite est relative par rapport à l'objet Composite.

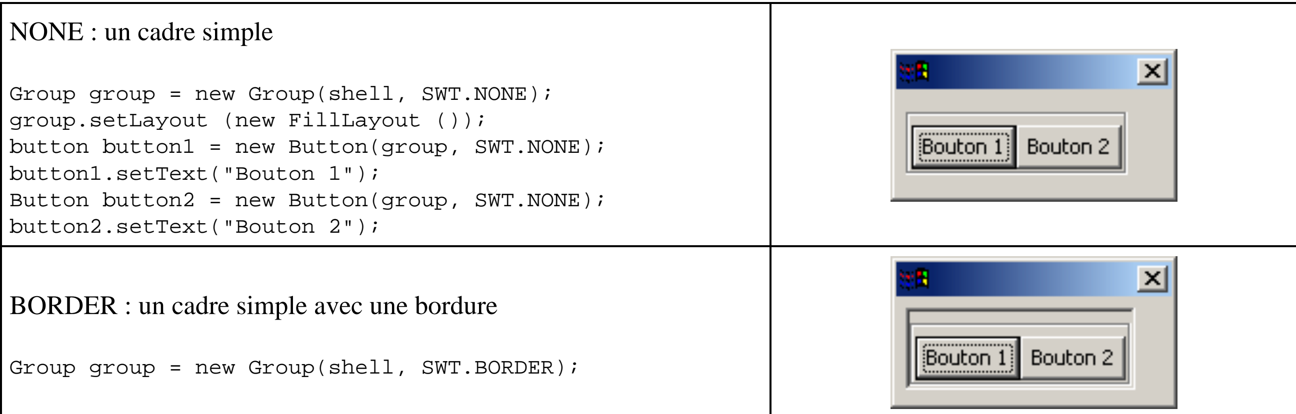
Exemple complet :

```
import org.eclipse.swt.*;
import org.eclipse.swt.graphics.*;
import org.eclipse.swt.widgets.*;

public class TestSWT2 {

  public static void main(String[] args) {
    Display display = new Display();
    Shell shell = new Shell(display);
    shell.setText("Test");

    Composite composite = new Composite(shell, SWT.BORDER);
    Color couleur = new Color(display,131,133,131);
    composite.setBackground(couleur);
    Label label = new Label(composite, SWT.NONE);
    label.setBackground(couleur);
    label.setText("Saisir la valeur");
    label.setBounds(10, 10, 100, 25);
    Text text = new Text(composite, SWT.BORDER);
    text.setText("mon texte");
    text.setBounds(10, 30, 100, 25);
    Button button = new Button(composite, SWT.BORDER);
    button.setText("Valider");
    button.setBounds(10, 60, 100, 25);
    composite.setSize(140,140);

    shell.pack();
    shell.open();
    while (!shell.isDisposed())
      if (!display.readAndDispatch())
        display.sleep();

    couleur.dispose();
    display.dispose();
  }
}
```

16.7.1.2. La classe Group

Ce contrôle permet de regrouper d'autres contrôles en les entourant d'une bordure et éventuellement d'un libellé.

La classe Group possède plusieurs styles : BORDER, SHADOW_ETCHED_IN, SHADOW_ETCHED_OUT, SHADOW_IN, SHADOW_OUT, SHADOW_NONE



NONE : un cadre simple

```
Group group = new Group(shell, SWT.NONE);
group.setLayout (new FillLayout ());
button button1 = new Button(group, SWT.NONE);
button1.setText("Bouton 1");
Button button2 = new Button(group, SWT.NONE);
button2.setText("Bouton 2");
```

BORDER : un cadre simple avec une bordure

```
Group group = new Group(shell, SWT.BORDER);
```

La méthode setText() permet de préciser un titre affiché en haut à gauche du cadre.

```
Group group = new Group(shell, SWT.NONE);
group.setLayout (new FillLayout ());
group.setText ("Actions");
Button button1 = new Button(group, SWT.NONE);
button1.setText("Bouton 1");
Button button2 = new Button(group, SWT.NONE);
button2.setText("Bouton 2");
```

16.7.2. Les contrôles de type « barre d'outils »

SWT permet de créer des barres d'outils fixes et barres d'outils flottantes.

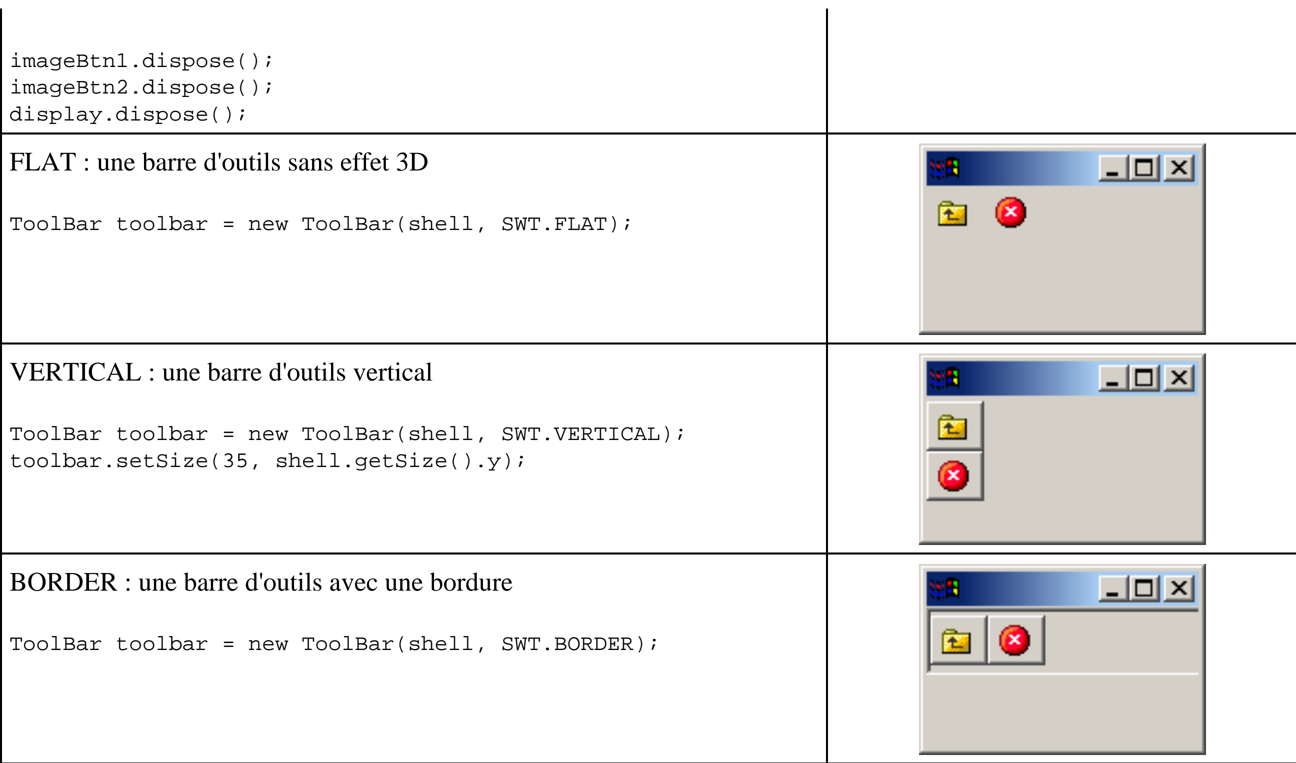
16.7.2.1. La classe ToolBar

Ce contrôle est une barre d'outils

La classe ToolBar possède plusieurs styles : BORDER, FLAT, WRAP, RIGHT, SHADOW_OUT HORIZONTAL, VERTICAL

HORIZONTAL : une barre d'outils horizontale (style par défaut)

```
shell.setSize(150, 100);
ToolBar toolbar = new ToolBar(shell, SWT.HORIZONTAL);
toolbar.setSize(shell.getSize().x, 35);
toolbar.setLocation(0, 0);

Image imageBtn1 = new Image(display, "btn1.bmp");
ToolItem btn1 = new ToolItem(toolbar, SWT.PUSH);
btn1.setImage(imageBtn1);

Image imageBtn2 = new Image(display, "btn2.bmp");
ToolItem btn2 = new ToolItem(toolbar, SWT.PUSH);
btn2.setImage(imageBtn2);

shell.open();
while (!shell.isDisposed())
  if (!display.readAndDispatch())
    display.sleep();
```



```
imageBtn1.dispose();
imageBtn2.dispose();
display.dispose();
```

FLAT : une barre d'outils sans effet 3D

```
ToolBar toolbar = new ToolBar(shell, SWT.FLAT);
```

VERTICAL : une barre d'outils vertical

```
ToolBar toolbar = new ToolBar(shell, SWT.VERTICAL);
toolbar.setSize(35, shell.getSize().y);
```

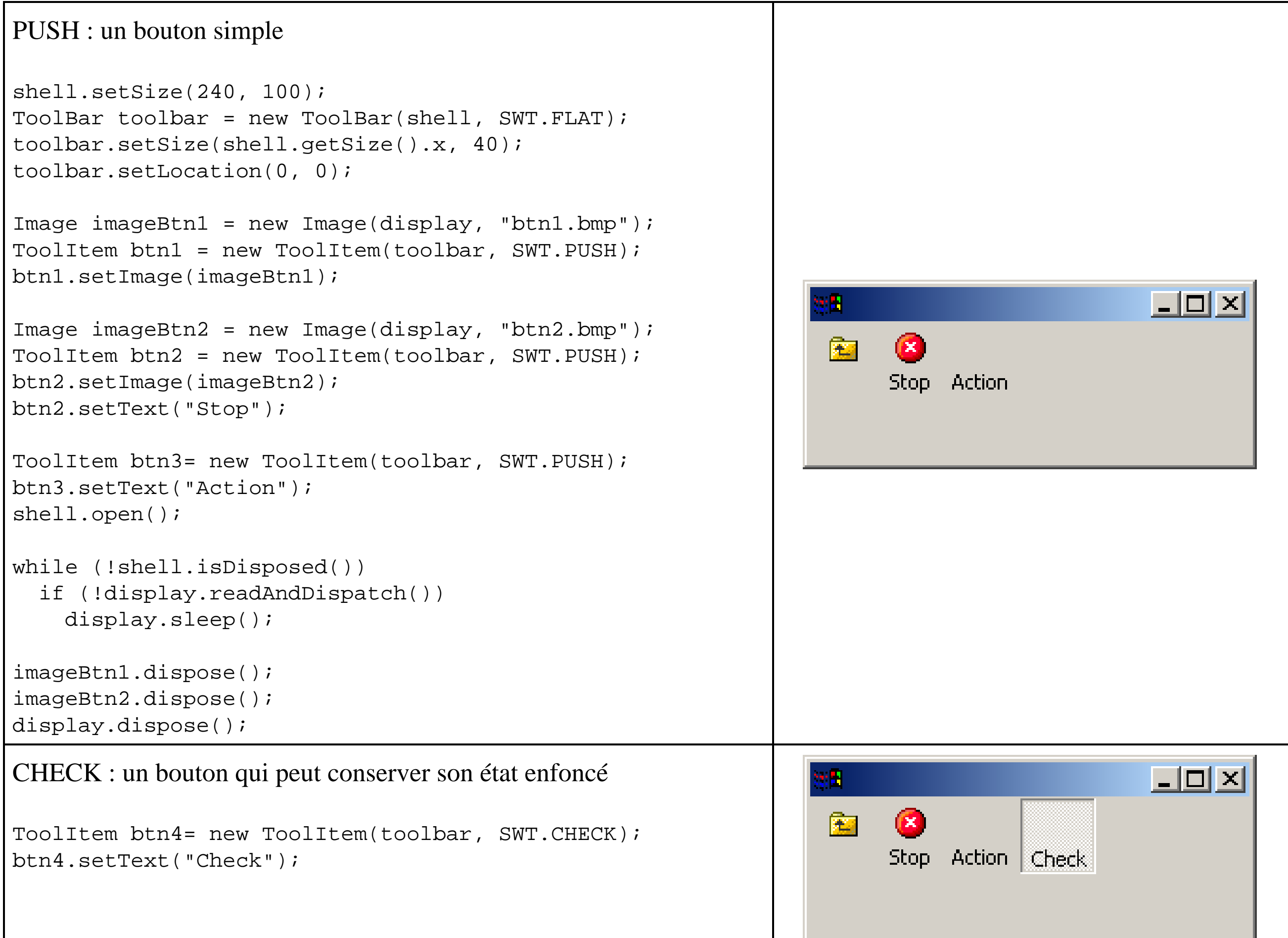
BORDER : une barre d'outils avec une bordure

```
ToolBar toolbar = new ToolBar(shell, SWT.BORDER);
```

16.7.2.2. La classe ToolItem

Ce contrôle est un élément d'une barre d'outils

La classe ToolItem possède styles : PUSH, CHECK, RADIO, SEPARATOR, DROP_DOWN



PUSH : un bouton simple

```
shell.setSize(240, 100);
ToolBar toolbar = new ToolBar(shell, SWT.FLAT);
toolbar.setSize(shell.getSize().x, 40);
toolbar.setLocation(0, 0);

Image imageBtn1 = new Image(display, "btn1.bmp");
ToolItem btn1 = new ToolItem(toolbar, SWT.PUSH);
btn1.setImage(imageBtn1);

Image imageBtn2 = new Image(display, "btn2.bmp");
ToolItem btn2 = new ToolItem(toolbar, SWT.PUSH);
btn2.setImage(imageBtn2);
btn2.setText("Stop");

ToolItem btn3= new ToolItem(toolbar, SWT.PUSH);
btn3.setText("Action");
shell.open();

while (!shell.isDisposed())
  if (!display.readAndDispatch())
    display.sleep();

imageBtn1.dispose();
imageBtn2.dispose();
display.dispose();
```

CHECK : un bouton qui peut conserver son état enfoncé

```
ToolItem btn4= new ToolItem(toolbar, SWT.CHECK);
btn4.setText("Check");
```

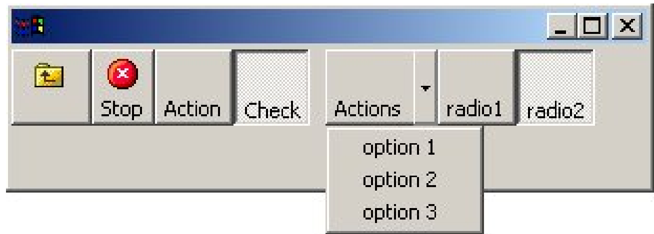
SEPARATOR : un séparateur `ToolItem btn5= new ToolItem(toolbar, SWT.SEPARATOR);`	
DROP_DOWN : un bouton avec une petite flèche vers le bas	

RADIO : un bouton dont un seul d'un même ensemble peut être sélectionné (un ensemble est défini par des boutons de type radio qui sont adjacents)
 	
`ToolItem btn7= new ToolItem(toolbar, SWT.RADIO);`
`btn7.setText("radio1");`
`ToolItem btn8= new ToolItem(toolbar, SWT.RADIO);`
`btn8.setText("radio2");`	

Exemple :

```
shell.setSize(340, 100);
final ToolBar toolbar = new ToolBar(shell, SWT.HORIZONTAL);
toolbar.setSize(shell.getSize().x, 45);
toolbar.setLocation(0, 0);

Image imageBtn1 = new Image(display, "btn1.bmp");
ToolItem btn1 = new ToolItem(toolbar, SWT.PUSH);
btn1.setImage(imageBtn1);

Image imageBtn2 = new Image(display, "btn2.bmp");
ToolItem btn2 = new ToolItem(toolbar, SWT.PUSH);
btn2.setImage(imageBtn2);
btn2.setText("Stop");

ToolItem btn3 = new ToolItem(toolbar, SWT.PUSH);
btn3.setText("Action");

ToolItem btn4 = new ToolItem(toolbar, SWT.CHECK);
btn4.setText("Check");

ToolItem btn5 = new ToolItem(toolbar, SWT.SEPARATOR);

final ToolItem btn6 = new ToolItem(toolbar, SWT.DROP_DOWN);
btn6.setText("Actions");

final Menu menu = new Menu(shell, SWT.POP_UP);
MenuItem menu1 = new MenuItem(menu, SWT.PUSH);
menu1.setText("option 1");
MenuItem menu2 = new MenuItem(menu, SWT.PUSH);
menu2.setText("option 2");
MenuItem menu3 = new MenuItem(menu, SWT.PUSH);
menu3.setText("option 3");

btn6.addListener(SWT.Selection, new Listener() {
  public void handleEvent(Event event) {
    if (event.detail == SWT.ARROW) {
      Rectangle rect = btn6.getBounds();
      Point pt = new Point(rect.x, rect.y + rect.height);
      pt = toolbar.toDisplay(pt);
      menu.setLocation(pt.x, pt.y);
      menu.setVisible(true);
    }
  }
});
```

```
ToolItem btn7 = new ToolItem(toolbar, SWT.RADIO);
btn7.setText("radio1");

ToolItem btn8 = new ToolItem(toolbar, SWT.RADIO);
btn8.setText("radio2");

shell.open();
while (!shell.isDisposed())
  if (!display.readAndDispatch())
    display.sleep();

imageBtn1.dispose();
imageBtn2.dispose();
display.dispose();
```



16.7.2.3. Les classes CoolBar et CoolItem

La classe CoolBar est un élément d'une barre d'outils repositionnable. Ce contrôle doit être utilisé avec un ou plusieurs contrôles CoolItem qui représentent un élément de la barre.

La classe CoolItem est un élément d'une barre de type CoolBar

Le plus simple est d'associer une barre d'outils de type ToolBar à un de ces éléments en utilisant la méthode setControl() de la classe CoolItem.

Exemple : utilisation de la barre d'outils définie dans la section précédente

```
shell.setLayout(new GridLayout());

shell.setSize(340, 100);
CoolBar coolbar = new CoolBar(shell, SWT.BORDER);
final ToolBar toolbar = new ToolBar(coolbar, SWT.FLAT);

Image imageBtn1 = new Image(display, "btn1.bmp");
ToolItem btn1 = new ToolItem(toolbar, SWT.PUSH);
btn1.setImage(imageBtn1);

Image imageBtn2 = new Image(display, "btn2.bmp");
ToolItem btn2 = new ToolItem(toolbar, SWT.PUSH);
btn2.setImage(imageBtn2);
btn2.setText("Stop");

ToolItem btn3 = new ToolItem(toolbar,SWT.PUSH);
btn3.setText("Action");

ToolItem btn4 = new ToolItem(toolbar,SWT.CHECK);
btn4.setText("Check");

ToolItem btn5 = new ToolItem(toolbar, SWT.SEPARATOR);

final ToolItem btn6 = new ToolItem(toolbar, SWT.DROP_DOWN);
btn6.setText("Actions");

final Menu menu = new Menu(shell, SWT.POP_UP);
MenuItem menu1 = new MenuItem(menu, SWT.PUSH);
menu1.setText("option 1");
MenuItem menu2 = new MenuItem(menu, SWT.PUSH);
```

```
menu2.setText("option2");
MenuItem menu3 = new MenuItem(menu, SWT.PUSH);
menu3.setText("option3");

btn6.addListener(SWT.Selection, new Listener() {
  public void handleEvent(Event event) {
    if (event.detail == SWT.ARROW) {
      Rectangle rect = btn6.getBounds();
      Point pt = new Point(rect.x, rect.y + rect.height);
      pt = toolbar.toDisplay(pt);
      menu.setLocation(pt.x, pt.y);
      menu.setVisible(true);
    }

  }
});

ToolItem btn7 = new ToolItem(toolbar, SWT.RADIO);
btn7.setText("radio1");

ToolItem btn8 = new ToolItem(toolbar, SWT.RADIO);
btn8.setText("radio2");

CoolItem coolItem = new CoolItem(coolbar, SWT.NONE);
coolItem.setControl(toolbar);
Point size = toolbar.computeSize(SWT.DEFAULT, SWT.DEFAULT);
coolItem.setPreferredSize(coolItem.computeSize(size.x, size.y));

shell.open();

while (!shell.isDisposed())
  if (!display.readAndDispatch())
    display.sleep();

imageBtn1.dispose();
imageBtn2.dispose();
```

La classe CoolItem possède une méthode setLocked() qui attend un booléen en paramètre précisant si le contrôle peut être déplacé ou non. Cette méthode doit être appelée lors d'un clic sur un bouton de la barre pour empêcher le déplacement de celle ci lors du clic sur le bouton.

16.8. La gestion des erreurs

Lors de l'utilisation de l'API SWT, des exceptions de trois types peuvent être levées :

- IllegalArgumentException : un argument fourni à une méthode est invalide
- SWTException : cette exception est levée lors d'une erreur non fatale. Le code de l'erreur et le message de l'exception permettent d'obtenir des précisions sur l'exception
- SWTError : cette exception est levée lors d'une erreur fatale

16.9. Le positionnement des contrôles

16.9.1. Le positionnement absolu

Dans ce mode, il faut préciser pour chaque composant, sa position et sa taille. L'inconvénient de ce mode de positionnement est qu'il réagit très mal à un changement de la taille du conteneur des composants.

16.9.2. Le positionnement relatif avec les LayoutManager

SWT propose un certain nombre de gestionnaires de positionnement de contrôles (layout manager). Ceux ci sont regroupés dans le package org.eclipse.swt.layout.

Le grand avantage de ce mode de positionnement est de laisser au LayoutManager utilisé le soin de positionner et de dimensionner chaque composant en fonction de ces règles et des paramètres qui lui sont fournis.

SWT définit quatre gestionnaires de positionnement :

- RowLayout pour un arrangement simple de modèle de mot-sur-un-page des composants
- FillLayout pour les composants égal-classés qui occupent une colonne ou une rangée simple
- GridLayout pour une grille rectangulaire des cellules composantes
- FormLayout pour un positionnement plus précis mais aussi plus compliqué des composants.

Pour personnaliser finement l'arrangement des composants, des informations complémentaires peuvent être associées à chacun d'eux en utilisant un objet dédié du type RowData, GridData ou FormData respectivement pour les gestionnaires de positionnement RowLayout, GridLayout et FormLayout.

16.9.2.1. FillLayout

Le FillLayout est le gestionnaire de positionnement le plus simple : il organise les composants dans une colonne ou une rangée. L'espace entre les composants est calculé automatiquement par la classe FillLayout.

La classe FillLayout peut utiliser deux styles : SWT.HORIZONTAL (par défaut) et SWT.VERTICAL pour préciser le mode d'alignement

Exemple :

```
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.*;

public class TestSWT27 {

  public static void main(String[] args) {
    final Display display = new Display();
    final Shell shell = new Shell(display);

    shell.setLayout(new RowLayout());
    Button bouton1 = new Button(shell, SWT.FLAT);
    bouton1.setText("bouton 1");
    Button bouton2 = new Button(shell, SWT.FLAT);
    bouton2.setText("bouton 2");
    Button bouton3 = new Button(shell, SWT.FLAT);
    bouton3.setText("bouton 3");

    shell.pack();
    shell.open();

    while (!shell.isDisposed()) {
      if (!display.readAndDispatch())
        display.sleep();
    }

    display.dispose();
  }
```

```
}
```

Voici différents aperçus en cas de modification de la taille de la fenêtre.

Il n'est pas possible de mettre un espace entre le bord du conteneur et les composants avec ce gestionnaire. Il n'est pas non plus possible pour ce gestionnaire de mettre des composants sur plusieurs colonnes ou rangées.

16.9.2.2. RowLayout

Ce gestionnaire propose d'arranger les composants en rangée horizontale ou verticale. Il possède des paramètres permettant de préciser une marge, un espace, une rupture et une compression.

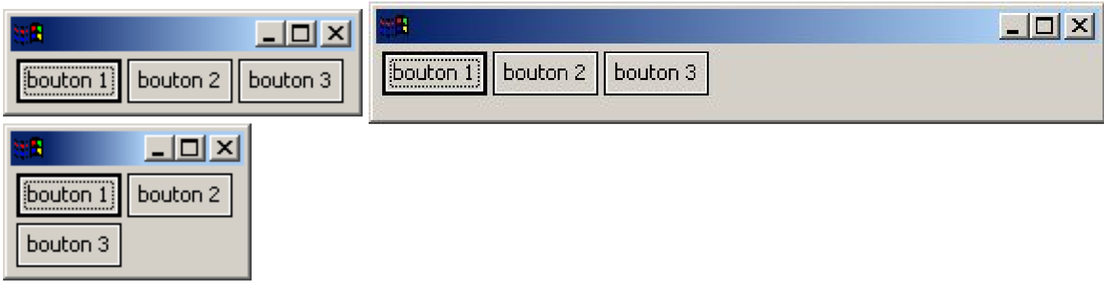
Propriété	Valeur par défaut	Rôle
wrap	true	demande de faire une rupture dans la rangée s'il n'y a plus de place false : true :
pack	true	demande à chaque composant de prendre sa taille préférée
justify	false	justification des composants
type	SWT.HORIZONTAL	type de mise en forme SWT.HORIZONTAL ou SWT.VERTICAL SWT.VERTICAL :
marginLeft	3	taille en pixel de la marge gauche
marginTop	3	taille en pixel de la marge haute
marginRight	3	taille en pixel de la marge droite
marginBottom	3	taille en pixel de la marge basse
spacing	3	taille en pixel entre deux cellules

Exemple :

```
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.layout.*;
```

```
import org.eclipse.swt.*;

public class TestSWT27 {

  public static void main(String[] args) {
    final Display display = new Display();
    final Shell shell = new Shell(display);

    RowLayout rowlayout = new RowLayout();
    shell.setLayout(rowlayout);
    Button bouton1 = new Button(shell, SWT.FLAT);
    bouton1.setText("bouton 1");

    Button bouton2 = new Button(shell, SWT.FLAT);
    bouton2.setText("bouton 2");

    Button bouton3 = new Button(shell, SWT.FLAT);
    bouton3.setText("bouton 3");

    shell.pack();
    shell.open();

    while (!shell.isDisposed()) {
      if (!display.readAndDispatch())
        display.sleep();
    }
    display.dispose();
  }
}
```

Voici différents aperçu en cas de modification de la taille de la fenêtre.



16.9.2.3. GridLayout

Ce gestionnaire permet d'arranger les composants dans une grille.

Ce gestionnaire possède plusieurs propriétés :

Propriété	Valeur par défaut	Rôle
horizontalSpacing	5	préciser l'espace horizontal entre chaque cellule
makeColumnsEqualWidth	false	donner à toute la colonne de la grille la même largeur
marginHeight	5	préciser la hauteur de la marge
marginWidth	5	préciser la largeur de la marge
numColumns	1	préciser le nombre de colonnes de la grille
verticalSpacing	5	préciser l'espace vertical entre chaque cellule

Exemple :

```
import org.eclipse.swt.widgets.*;
```

```
import org.eclipse.swt.layout.*;
import org.eclipse.swt.*;

public class TestSWT28 {

  public static void main(String[] args) {
    final Display display = new Display();
    final Shell shell = new Shell(display);

    GridLayout gridLayout = new GridLayout();
    gridLayout.numColumns = 2;
    shell.setLayout(gridLayout);

    Label label1 = new Label(shell, SWT.NONE);
    label1.setText("Donnee 1 :");
    Text text1 = new Text(shell, SWT.BORDER);
    text1.setSize(200, 10);

    Label label2 = new Label(shell, SWT.NONE);
    label2.setText("Donnee 2:");
    Text text2 = new Text(shell, SWT.BORDER);
    text2.setSize(200, 10);

    Label label3 = new Label(shell, SWT.NONE);
    label3.setText("Donnee 3 :");
    Text text3 = new Text(shell, SWT.BORDER);
    text3.setSize(200, 10);

    Button button1 = new Button(shell, SWT.NONE);
    button1.setText("Valider");

    Button button2 = new Button(shell, SWT.NONE);
    button2.setText("Annuler");

    shell.pack();
    shell.open();

    while (!shell.isDisposed()) {
      if (!display.readAndDispatch())
        display.sleep();
    }

    display.dispose();
  }

}
```

Les paramètres liés à un composant d'une cellule particulière de la grille peuvent être précisés grâce à un objet de type GridData. Ces paramètres précisent le comportement du composant en cas de redimensionnement.

Il existe deux façons de créer un objet de type GridData :

- instancier un objet de type GridData avec son constructeur sans paramètre et initialiser les propriétés en utilisant les setters appropriés.
- instancier un objet de type GridData avec son constructeur attendant un style en paramètre

La méthode setLayoutData() permet de d'associer un objet GridData à un composant.

Attention : il ne faut pas utiliser plusieurs fois un objet de type GridData.

Exemple :

```
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.*;

public class TestSWT28 {

  public static void main(String[] args) {
    final Display display = new Display();
    final Shell shell = new Shell(display);

    GridLayout gridLayout = new GridLayout();
    gridLayout.numColumns = 2;
    shell.setLayout(gridLayout);

    Label label1 = new Label(shell, SWT.NONE);
    label1.setText("Donnee 1 :");
    Text text1 = new Text(shell, SWT.BORDER);
    text1.setSize(200, 10);

    Label label2 = new Label(shell, SWT.NONE);
    label2.setText("Donnee 2:");
    Text text2 = new Text(shell, SWT.BORDER);
    text2.setSize(200, 10);

    Label label3 = new Label(shell, SWT.NONE);
    label3.setText("Donnee 3 :");
    Text text3 = new Text(shell, SWT.BORDER);
    text3.setSize(200, 10);

    Button button1 = new Button(shell, SWT.NONE);

    button1.setText("Valider");

    Button button2 = new Button(shell, SWT.NONE);
    button2.setText("Annuler");

    GridData data = new GridData();
    data.widthHint = 120;
    label1.setLayoutData(data);

    data = new GridData();
    data.widthHint = 220;
    text1.setLayoutData(data);

    shell.pack();
    shell.open();

    while (!shell.isDisposed()) {
      if (!display.readAndDispatch())
        display.sleep();
    }

    display.dispose();
  }

}
```

16.9.2.4. FormLayout

Ce gestionnaire possède deux propriétés :

Propriété	Valeur par défaut	Rôle
marginHeight	0	préciser la hauteur de la marge
marginWidth	0	préciser la largeur de la marge

Ce gestionnaire impose d'associer à chaque composant un objet de type FormData qui va préciser les informations de positionnement et de comportement du composant.

16.10. La gestion des événements

La gestion des événements avec SWT est très similaire à celle proposée par l'API Swing car elle repose sur les Listeners. Ces Listeners doivent être ajoutés au contrôle en fonction des événements qu'ils doivent traiter.

Dès lors, lorsque l'événement est émis suite à une action de l'utilisateur, la méthode correspondante du Listener enregistré est exécutée.

Dans la pratique, les Listeners sont des interfaces qu'il faut faire implémenter par une classe selon les besoins. Cette implémentation définira donc des méthodes qui contiennent les traitements à exécuter pour un événement précis. Un ou plusieurs paramètres fournis à ces méthodes permettent d'obtenir des informations plus précises sur l'événement.

Il suffit ensuite d'enregistrer le Listener auprès du contrôle en utilisant la méthode addXXXListener() du contrôle ou XXX représente le type du Listener.

Exemple : pour le traitement d'un clic d'un bouton

```
import org.eclipse.swt.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.widgets.*;

public class TestSWT3 {

  public static void main(String[] args) {
    Display display = new Display();
    Shell shell = new Shell(display);
    shell.setText("Test");

    Button button = new Button(shell, SWT.NONE);
    button.setText("Valider");
    button.setBounds(1, 1, 100, 25);

    button.addSelectionListener(new SelectionListener() {
      public void widgetSelected(SelectionEvent arg0) {
                        System.out.println("Appui sur le bouton");
      }
      public void widgetDefaultSelected(SelectionEvent arg0) {
      }
    });

    shell.pack();
    shell.open();

    while (!shell.isDisposed())
      if (!display.readAndDispatch())
        display.sleep();

    display.dispose();
  }
}
```

Comme avec Swing, SWT propose un ensemble de classe de type Adapter qui sont des classes qui implémentent une des interfaces Listeners avec toutes ces méthodes vides. Pour les utiliser, il suffit de définir une classe fille qui hérite de la classe de type Adapter adéquat et de redéfinir la ou les méthodes utiles.

Exemple :

```
import org.eclipse.swt.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.widgets.*;

public class TestSWT4 {

  public static void main(String[] args) {
    Display display = new Display();
    Shell shell = new Shell(display);
    shell.setText("Test");

    Button button = new Button(shell, SWT.NONE);
    button.setText("Valider");
    button.setBounds(1, 1, 100, 25);

    button.addSelectionListener(new SelectionAdapter() {
      public void widgetSelected(SelectionEvent arg0) {
        System.out.println("Appui sur le bouton");
      }
    });

    shell.pack();
    shell.open();

    while (!shell.isDisposed())
      if (!display.readAndDispatch())
        display.sleep();

    display.dispose();
  }
}
```

SWT définit plusieurs Listeners :

- SelectionListener : événement lié à la sélection d'un élément du contrôle
- KeyListener : événement lié au clavier
- MouseListener : événement lié aux clics de la souris
- MouseMoveListener : événement lié au mouvement de la souris
- MouseTrackListener : événement lié à la souris par rapport au contrôle (entrée, sortie, passage au dessus)
- ModifyListerner : événement lié à la modification du contenu d'un contrôle de saisie de texte
- VerifyListener : événement lié à la vérification avant modification du contenu d'un contrôle de saisie de texte
- FocusListener : événement lié à prise ou à la perte du focus
- TraverseListener : événement lié à la traversée d'un contrôle au moyen de la touche tab ou des flèches
- PaintListener : événement lié à nécessité de redessiner le composant

16.10.1. L'interface KeyListener

Cette interface définit deux méthodes keyPressed() et keyReleased() relatives à des événements émis par le clavier, respectivement l'enfoncement d'une touche et la relâche d'une touche du clavier.

Ces deux méthodes possèdent un objet de type KeyEvent qui contient des informations sur l'événement grâce à trois attributs :

character	contient le caractère la touche concernée

keyCode	contient le code de la touche concernée SWT défini des valeurs pour des touches particulières, par exemple SWT.ALT, SWT.ARROW_DOWN, SWT_ARROW_LEFT, SWT.CTRL, SWT.CR, SWT.F1, SWT.F2, ...
stateMask	contient l'état du clavier au moment de l'émission de l'événement, ce qui permet de savoir par exemple si la touche Alt ou Shift ou Ctrl est enfoncée au moment de l'événement en effectuant un test sur la valeur avec SWT.ALT ou SWT.CTRL ou SWT.SHIFT

SWT définit une classe KeyAdapter qui implémente l'interface KeyListener avec des méthodes vides.

Exemple :

```
import org.eclipse.swt.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.widgets.*;

public class TestSWT5 {

  public static void main(String[] args) {
    Display display = new Display();
    Shell shell = new Shell(display);
    shell.setText("Test");

    shell.addKeyListener(new KeyAdapter() {
      public void keyReleased(KeyEvent e) {
        String res = "";
        switch (e.character) {
          case SWT.CR :
            res = "Touche Entree";
            break;
          case SWT.DEL :
            res = "Touche Supp";
            break;
          case SWT.ESC :
            res = "Touche Echap";
            break;
          default :
            res = res + e.character;
        }

        System.out.println(res);

      }
    });

    shell.pack();
    shell.open();

    while (!shell.isDisposed())
      if (!display.readAndDispatch())
        display.sleep();

    display.dispose();
  }
}
```

Exemple : utilisation de la propriété stateMask

```
import org.eclipse.swt.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.widgets.*;

public class TestSWT6 {

  public static void main(String[] args) {
    Display display = new Display();
    Shell shell = new Shell(display);
    shell.setText("Test");
```

```
    shell.addKeyListener(new KeyAdapter() {
      public void keyReleased(KeyEvent e) {
        String res = "";
        if (e.keyCode == SWT.SHIFT) {
            res = "touche shift";
        } else {
          if ((e.stateMask & SWT.SHIFT) != 0) {
            res = "" + e.character + " + touche shift";
          }
        }
        System.out.println(res);
      }
    });

    shell.pack();
    shell.open();

    while (!shell.isDisposed())
      if (!display.readAndDispatch())
        display.sleep();

    display.dispose();
  }
}
```

16.10.2. L'interface MouseListener

Cette interface définit trois méthodes mouseDown(), mouseUp() et mouseDoubleClick() relative à des événements émis par un clic sur la souris, respectivement l'enfoncement d'un bouton et le relâchement d'un bouton ou le double clic sur un bouton de la souris.

Ces trois méthodes possèdent un objet de type MouseEvent qui contient des informations sur l'événement grâce à quatre attributs :

button	contient le numéro du bouton utilisé (de 1 à 3). Attention, la valeur est dépendante du système utilisé, par exemple sous Windows avec une souris à molette possédant deux boutons, l'appui sur le bouton de droite renvoie 3
stateMask	contient l'état du clavier au moment de l'émission de l'événement, ce qui permet de savoir par exemple si la touche Alt ou Shift ou Ctrl est enfoncée au moment de l'événement en effectuant un test sur la valeur avec SWT.ALT ou SWT.CTRL ou SWT.SHIFT
x	contient la coordonnée x du pointeur de la souris par rapport au contrôle lors de l'émission de l'événement
y	contient la coordonnée y du pointeur de la souris par rapport au contrôle lors de l'émission de l'événement

SWT définit une classe MouseAdapter qui implémente l'interface MouseListener avec des méthodes vides.

Exemple :

```
import org.eclipse.swt.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.widgets.*;

public class TestSWT7 {

  public static void main(String[] args) {
    Display display = new Display();
    Shell shell = new Shell(display);
    shell.setText("Test");
```

```
    shell.addMouseListener(new MouseAdapter() {
      public void mouseDown(MouseEvent e) {
        String res = "";
          res = "bouton " + e.button + ", x = " + e.x + ", y = " + e.y;
          if ((e.stateMask & SWT.SHIFT) != 0) {
            res = res + " + touche shift";
          }
        System.out.println(res);
      }
    });

    shell.pack();
    shell.open();

    while (!shell.isDisposed())
      if (!display.readAndDispatch())
        display.sleep();

    display.dispose();
  }
}
```

16.10.3. L'interface MouseMoveListener

Cette interface définit une seule méthode mouseMove() relative à un événement émis par un déplacement de la souris au dessus d'un contrôle.

Cette méthode possède un objet de type MouseEvent qui contient des informations sur l'événement grâce à quatre attributs.

La mise en oeuvre est similaire de l'interface MouseListener.

Exemple :

```
import org.eclipse.swt.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.widgets.*;

public class TestSWT8 {

  public static void main(String[] args) {
    Display display = new Display();
    Shell shell = new Shell(display);
    shell.setText("Test");

    shell.addMouseMoveListener(new MouseMoveListener() {
      public void mouseMove(MouseEvent e) {
        String res = "";
        if ((e.x < 20) & (e.y < 20)) {

          res = "x = " + e.x + ", y = " + e.y;
          if ((e.stateMask & SWT.SHIFT) != 0) {
            res = res + " + touche shift";
          }
          System.out.println(res);
        }
      }
    });

    shell.pack();
    shell.open();

    while (!shell.isDisposed())
      if (!display.readAndDispatch())
        display.sleep();
```

```
    display.dispose();
  }
}
```

16.10.4. L'interface MouseTrackListener

Cette interface définit trois méthodes mouseEnter(), MouseExit() et mouseHover() relative à des événements émis respectivement par l'entrée de la souris sur la zone d'un composant, la sortie et le passage au dessus de la zone d'un composant.

Ces trois méthodes possèdent un objet de type MouseEvent.

SWT définit une classe MouseTrackAdapter qui implémente l'interface MouseListener avec des méthodes vides.

Exemple : changement de la couleur de fond de la fenêtre

```
import org.eclipse.swt.events.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.graphics.*;

public class TestSWT9 {

  public static void main(String[] args) {
    Display display = new Display();
    final Shell shell = new Shell(display);

    final Color couleur1 = new Color(display,155,130,0);
    final Color couleur2 = new Color(display,130,130,130);
    shell.setText("Test");

    shell.addMouseTrackListener(new MouseTrackAdapter() {
      public void mouseEnter(MouseEvent e) {
            shell.setBackground(couleur1);
      }
      public void mouseExit(MouseEvent e) {
            shell.setBackground(couleur2);
      }
    });

    shell.pack();
    shell.open();

    while (!shell.isDisposed())
      if (!display.readAndDispatch())
        display.sleep();

    display.dispose();
  }
}
```

16.10.5. L'interface ModifyListener

Cette interface définit une seule méthode modifyText() relative à un événement émis lors de la modification du contenu d'un contrôle de saisie de texte.

Cette méthode possède un objet de type ModifyEvent.

Exemple :

```
import org.eclipse.swt.*;
import org.eclipse.swt.events.*;
```

```
import org.eclipse.swt.widgets.*;

public class TestSWT10 {

  public static void main(String[] args) {
    Display display = new Display();
    Shell shell = new Shell(display);
    shell.setText("Test");

    Text text = new Text(shell, SWT.BORDER);
    text.setText("mon texte");
    text.setSize(100, 25);

   text.addModifyListener(new ModifyListener() {
     public void modifyText(ModifyEvent e) {
       System.out.println("nouvelle valeur = " + ((Text)e.widget).getText());
      }
    });

    shell.pack();
    shell.open();

    while (!shell.isDisposed())
      if (!display.readAndDispatch())
        display.sleep();

    display.dispose();
  }
}
```

16.10.6. L'interface VerifyText()

Cette interface définit une seule méthode verifyText() relative à un événement émis lors de la vérification des données avant modification du contenu d'un contrôle de saisie de texte.

Cette méthode possède un objet de type VerifyEvent qui contient des informations sur l'événement grâce à quatre attributs :

doit	un drapeau qui indique si la modification doit être effectuée ou non
end	la position de début de la modification
start	la position de fin de la modification
text	la valeur de la modification

Exemple : n'autoriser la saisie que de chiffres

```
import org.eclipse.swt.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.widgets.*;

public class TestSWT11 {

  public static void main(String[] args) {
    Display display = new Display();
    Shell shell = new Shell(display);
    shell.setText("Test");

    Text text = new Text(shell, SWT.BORDER);
    text.setText("");
    text.setSize(100, 25);

    text.addVerifyListener(new VerifyListener() {
      public void verifyText(VerifyEvent e) {
        int valeur = 0;
        e.doit = true;
```

```
        if (e.text != "") {
          try {
            valeur = Integer.parseInt(e.text);
          } catch (NumberFormatException e1) {
            e.doit = false;
          }
        }

        System.out.println(
          "start = " + e.start + ", end = " + e.end + ", text = " + e.text);

      }
    });

    shell.pack();
    shell.open();

    while (!shell.isDisposed())
      if (!display.readAndDispatch())
        display.sleep();

    display.dispose();
  }
}
```

16.10.7. L'interface FocusListener

Cette interface définit deux méthodes focusGained() et focusLost() relative à un événement émis respectivement lors de la prise et la perte du focus par un contrôle.

Ces méthodes possèdent un objet de type FocusEvent.

SWT définit une classe FocusAdapter qui implémente l'interface FocusListener avec des méthodes vides.

Exemple :

```
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.*;

public class TestSWT12 {

  public static void main(String[] args) {
    Display display = new Display();
    Shell shell = new Shell(display);

    Text text = new Text(shell, SWT.BORDER);
    text.setText("mon texte");
    text.setBounds(10, 10, 100, 25);

    text.addFocusListener(new FocusListener() {
      public void focusGained(FocusEvent e) {
        System.out.println(e.widget + " obtient le focus");
      }
      public void focusLost(FocusEvent e) {
        System.out.println(e.widget + " perd le focus");
      }
    });

    Button button = new Button(shell, SWT.NONE);
    button.setText("Valider");
    button.setBounds(10, 40, 100, 25);

    shell.pack();
    shell.open();

    while (!shell.isDisposed())
```

```
      if (!display.readAndDispatch())
        display.sleep();

    display.dispose();
  }
}
```

16.10.8. L'interface TraverseListener

Cette interface définit une méthode keyTraversed() relative à un événement émis lors de la traversée d'un contrôle au moyen de la touche tab ou des flèches haut et bas.

Cette méthode possède un objet de type VerifyEvent qui contient des informations sur l'événement grâce à deux attributs :

doit	un drapeau qui indique si le composant peut être traversé ou non
detail	le type de l'opération qui génère la traversée

Exemple : empêcher le parcours des contrôles dans l'ordre inverse par la touche tab

```
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.*;

public class TestSWT13 {

  public static void main(String[] args) {
    Display display = new Display();
    Shell shell = new Shell(display);

    TraverseListener tl = new TraverseListener() {
      public void keyTraversed(TraverseEvent e) {
        String res = "";
        res = e.widget + " est traverse grace à ";
        switch (e.detail) {
          case SWT.TRAVERSE_TAB_NEXT :
            res = res + " l'appui sur la touche tab";
            e.doit = true;
            break;
          case SWT.TRAVERSE_TAB_PREVIOUS :
            res = res + " l'appui sur la touche shift + tab";
            e.doit = false;
            break;
          default :
            res = res + " un autre moyen";
        }
        System.out.println(res);
      }
    };

    Text text = new Text(shell, SWT.BORDER);
    text.setText("mon texte");
    text.setBounds(10, 10, 100, 25);
    text.addTraverseListener(tl);

    Button button = new Button(shell, SWT.NONE);
    button.setText("Valider");
    button.setBounds(10, 40, 100, 25);
    button.addTraverseListener(tl);

    shell.pack();
    shell.open();

    while (!shell.isDisposed())
      if (!display.readAndDispatch())
```

```
        display.sleep();

    display.dispose();
  }
}
```

16.10.9. L'interface PaintListener

Cette interface définit une méthode paintControl() relative à un événement émis lors de la nécessité de redessiner le composant.

Cette méthode possède un objet de type PaintEvent qui contient des informations sur l'événement grâce à plusieurs attributs :

gc	un objet de type GC qui encapsule le contexte graphique
height	la hauteur de la zone à redessiner
width	la longueur de la zone à redessiner
x	l'abscisse de la zone à redessiner
y	l'ordonnée de la zone à redessiner

Exemple :

```
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.graphics.*;

public class TestSWT21 {

  public static void main(String[] args) {
    final Display display = new Display();
    final Shell shell = new Shell(display);
    shell.setSize(420, 420);

    Canvas canvas = new Canvas(shell, SWT.NONE);
    canvas.setSize(200, 200);
    canvas.setLocation(10, 10);
    canvas.addPaintListener(new PaintListener() {
      public void paintControl(PaintEvent e) {
        GC gc = e.gc;
        gc.drawText("Bonjour", 20, 20);
        gc.drawLine(10, 10, 10, 100);
        gc.setForeground(display.getSystemColor(SWT.COLOR_RED));
        gc.drawOval(60, 60, 60, 60);
      }
    });

    shell.pack();
    shell.open();

    GC gc = new GC(canvas);

    while (!shell.isDisposed()) {
      if (!display.readAndDispatch())
        display.sleep();
    }

    display.dispose();
  }

}
```

Remarque : il est important d'associer le listener avant d'ouvrir la fenêtre. Il n'est pas utile d'utiliser la méthode dispose() de l'objet de type GC car le code n'est pas responsable de son instanciation.

16.11. Les boîtes de dialogue

16.11.1. Les boîtes de dialogues prédéfinies

SWT propose plusieurs boîtes de dialogue prédéfinies.

16.11.1.1. La classe MessageBox

La classe MessageBox permet d'afficher un message à l'utilisateur et éventuellement de sélectionner une action standard via un bouton.

Les styles utilisables avec MessageBox sont :

_ ICON_ERROR, ICON_INFORMATION, ICON_QUESTION, ICON_WARNING, ICON_WORKING pour sélectionner l'icône affichée dans la boîte de dialogue

_ OK ou OK | CANCEL : pour boîte avec des boutons de type « Ok » / « Annuler »

_ YES | NO, YES | NO | CANCEL : pour boîte avec des boutons de type « Oui » / « Non » / « Annuler »

_ RETRY | CANCEL : pour boîte avec des boutons de type « Réessayer » / « Annuler »

_ ABORT | RETRY | IGNORE : pour boîte avec des boutons de type « Abandon » / « Réessayer » / « Ignorer »

La méthode setMessage() permet de préciser le message qui va être affiché à l'utilisateur.

La méthode open permet d'ouvrir la boîte de dialogue et de connaître le bouton qui à été utilisé pour fermer la boîte de dialogue en comparant la valeur de retour avec la valeur de style du bouton correspondant.

Exemple :

```
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.layout.*;

public class TestSWT18 {

  public static void main(String[] args) {
    final Display display = new Display();
    final Shell shell = new Shell(display);
    shell.setLayout(new GridLayout());
    shell.setSize(300, 300);

    Button btnOuvrir = new Button(shell, SWT.PUSH);
    btnOuvrir.setText("Afficher");
    btnOuvrir.addListener(SWT.Selection, new Listener() {
      public void handleEvent(Event e) {
        int reponse = 0;
        MessageBox mb = new MessageBox(shell,
          SWT.ICON_INFORMATION | SWT.ABORT | SWT.RETRY | SWT.IGNORE);
        mb.setMessage("Message d'information pour l'utilisateur");
        reponse = mb.open();
        if (reponse == SWT.ABORT) {
          System.out.println("Bouton abandonner selectionne");
        }
        if (reponse == SWT.RETRY) {
```

```
          System.out.println("Bouton reessayer selectionne");
        }
        if (reponse == SWT.IGNORE) {
          System.out.println("Bouton ignorer selectionne");
        }
      }
    });

    shell.pack();
    shell.open();
    while (!shell.isDisposed()) {
      if (!display.readAndDispatch())
        display.sleep();
    }
    display.dispose();
  }

}
```

16.11.1.2. La classe ColorDialog

Cette boîte de dialogue permet la sélection d'une couleur dans la palette des couleurs.

La méthode setRGB() permet de préciser la couleur qui est sélectionnée par défaut.

La méthode open() permet d'ouvrir la boîte de dialogue et de renvoyer la valeur de la couleur sélectionné sous la forme d'un objet de type RGB. Si aucune couleur n'est sélectionnée (appui sur le bouton annuler dans la boîte de dialogue) alors l'objet renvoyé est null.

Exemple :

```
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.graphics.*;

public class TestSWT15 {

  public static void main(String[] args) {
    final Display display = new Display();
    final Shell shell = new Shell(display);
    shell.setLayout(new GridLayout());
    shell.setSize(300, 300);

    Button btnOuvrir = new Button(shell, SWT.PUSH);
    btnOuvrir.setText("Couleur");
    btnOuvrir.addListener(SWT.Selection, new Listener() {
      public void handleEvent(Event e) {
        Color couleurDeFond = shell.getBackground();

        ColorDialog colorDialog = new ColorDialog(shell);
        colorDialog.setRGB(couleurDeFond.getRGB());
        RGB couleur = colorDialog.open();

        if (couleur != null) {
          if (couleurDeFond != null)
            couleurDeFond.dispose();
          couleurDeFond = new Color(display, couleur);
          shell.setBackground(couleurDeFond);
        }

      }
    });

    shell.getBackground().dispose();
    shell.open();
    while (!shell.isDisposed()) {
```

```
      if (!display.readAndDispatch())
        display.sleep();
    }
    display.dispose();
  }

}
```

16.11.1.3. La classe FontDialog

Cette classe encapsule une boîte de dialogue permettant la sélection d'une police de caractère.

La méthode open() permet d'ouvrir la boîte de dialogue et renvoie un objet de type FontData qui encapsule les données de la police sélectionnée ou renvoie null si aucune n'a été sélectionnée.

Exemple :

```
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.graphics.*;
import org.eclipse.swt.layout.*;

public class TestSWT19 {

  public static void main(String[] args) {
    final Display display = new Display();
    final Shell shell = new Shell(display);
    shell.setLayout(new GridLayout());
    shell.setSize(300, 300);

    Label lblNomPolice = new Label(shell, SWT.NONE);
    lblNomPolice.setText("Nom de la police = ");
    final Text txtNomPolice = new Text(shell, SWT.BORDER | SWT.READ_ONLY);
    txtNomPolice.setText("");
    txtNomPolice.setSize(280, 40);

    Button btnOuvrir = new Button(shell, SWT.PUSH);
    btnOuvrir.setText("Police");
    btnOuvrir.addListener(SWT.Selection, new Listener() {
      public void handleEvent(Event e) {
        FontDialog dialog = new FontDialog(shell, SWT.OPEN);
        FontData fontData = dialog.open();
        if (fontData != null) {
          txtNomPolice.setText(fontData.getName());
          System.out.println("selection de la police " + fontData.getName());
          if (txtNomPolice.getFont() != null) {
            txtNomPolice.getFont().dispose();
          }
          Font font = new Font(display, fontData);
          txtNomPolice.setFont(font);

        }
      }
    });

    shell.pack();
    shell.open();
    while (!shell.isDisposed()) {
      if (!display.readAndDispatch())
        display.sleep();
    }
    display.dispose();
  }

}
```

16.11.1.4. La classe FileDialog

Cette boîte de dialogue permet de sélectionner un fichier.

La méthode open() ouvre la boîte de dialogue et renvoie le nom du fichier sélectionné. Si aucun fichier n'est sélectionné, alors elle renvoie null.

La méthode setFilterExtensions() permet de préciser sous la forme d'un tableau de chaîne la liste des extensions de fichier acceptés par la sélection.

Exemple :

```
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.layout.*;

public class TestSWT16 {

  public static void main(String[] args) {
    final Display display = new Display();
    final Shell shell = new Shell(display);
    shell.setLayout(new GridLayout());
    shell.setSize(300, 300);

    Label lblNomFichier = new Label(shell, SWT.NONE);
    lblNomFichier.setText("Nom du fichier = ");
            final Text txtNomFichier = new Text(shell, SWT.BORDER | SWT.READ_ONLY);
            txtNomFichier.setText("");
            txtNomFichier.setSize(280,40);


    Button btnOuvrir = new Button(shell, SWT.PUSH);
    btnOuvrir.setText("Ouvrir");
    btnOuvrir.addListener(SWT.Selection, new Listener() {
      public void handleEvent(Event e) {
        String nomFichier;
        FileDialog dialog = new FileDialog(shell, SWT.OPEN);
        dialog.setFilterExtensions(new String[] { "*.java", "*.*" });
        nomFichier = dialog.open();
        if ((nomFichier != null) && (nomFichier.length() != 0)){
          txtNomFichier.setText(nomFichier);
            System.out.println("selection du fichier "+nomFichier);
        }
      }
    });

    shell.pack();
    shell.open();
    while (!shell.isDisposed()) {
      if (!display.readAndDispatch())
        display.sleep();
    }

    display.dispose();
  }

}
```

16.11.1.5. La classe DirectoryDialog

Cette classe encapsule une boîte de dialogue qui permet la sélection d'un répertoire.

La méthode open() ouvre la boîte de dialogue et renvoie le nom du répertoire sélectionné. Si aucun répertoire n'est sélectionné, alors elle renvoie null.

La méthode setFilterPath () permet de préciser sous la forme d'une chaîne de caractère le répertoire sélectionné par défaut.

La méthode setMessage() permet de préciser un message qui sera affiché dans la boîte de dialogue.

Exemple :

```
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.layout.*;

public class TestSWT17 {

  public static void main(String[] args) {
    final Display display = new Display();
    final Shell shell = new Shell(display);
    shell.setLayout(new GridLayout());
    shell.setSize(300, 300);

    Label lblNomFichier = new Label(shell, SWT.NONE);
    lblNomFichier.setText("Nom du fichier = ");
            final Text txtNomRepertoire = new Text(shell, SWT.BORDER | SWT.READ_ONLY);
            txtNomRepertoire.setText("");
            txtNomRepertoire.setSize(280,40);


    Button btnOuvrir = new Button(shell, SWT.PUSH);
    btnOuvrir.setText("Ouvrir");
    btnOuvrir.addListener(SWT.Selection, new Listener() {
      public void handleEvent(Event e) {
        String nomRepertoire;
        DirectoryDialog dialog = new DirectoryDialog(shell, SWT.OPEN);
        dialog.setFilterPath("d:/");
        dialog.setMessage("Test");
        nomRepertoire = dialog.open();
        if ((nomRepertoire != null) && (nomRepertoire.length() != 0)){
          txtNomRepertoire.setText(nomRepertoire);
          System.out.println("selection du repertoire "+nomRepertoire);
        }
      }
    });

    shell.pack();
    shell.open();
    while (!shell.isDisposed()) {
      if (!display.readAndDispatch())
        display.sleep();
    }
    display.dispose();
  }

}
```

16.11.1.6. La classe PrintDialog

La classe PrintDialog encapsule une boîte de dialogue permettant la sélection d'une imprimante configurée sur le système. Pour utiliser classe, il faut importer la package org.eclipse.swt.printing.

La méthode open() permet d'ouvrir la boîte de dialogue et renvoie un objet de type PrinterData qui encapsule les données de l'imprimante sélectionnée ou renvoie null si aucune n'a été sélectionnée.

Exemple :

```
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.printing.*;
```

```
import org.eclipse.swt.graphics.*;

public class TestSWT20 {

  public static void main(String[] args) {
    final Display display = new Display();
    final Shell shell = new Shell(display);
    shell.setLayout(new GridLayout());
    shell.setSize(300, 300);

    Button btnOuvrir = new Button(shell, SWT.PUSH);
    btnOuvrir.setText("Imprimer");
    btnOuvrir.addListener(SWT.Selection, new Listener() {
      public void handleEvent(Event e) {
        PrintDialog dialog = new PrintDialog(shell, SWT.OPEN);
        PrinterData printerData = dialog.open();
        if (printerData != null) {
          Printer printer = new Printer(printerData);
          if (printer.startJob("Test")) {
            printer.startPage();
            GC gc = new GC(printer);
            gc.drawString("Bonjour", 100, 100);
            printer.endPage();
            printer.endJob();
            gc.dispose();
            printer.dispose();
          }
        }
      }
    });

    shell.pack();
    shell.open();
    while (!shell.isDisposed()) {
      if (!display.readAndDispatch())
        display.sleep();
    }
    display.dispose();
  }

}
```

Remarque : cet exemple est très basic dans la mesure où il est préférable de lancer les taches d'impression dans un thread pour ne pas bloquer l'interface utilisateur pendant ces traitements.

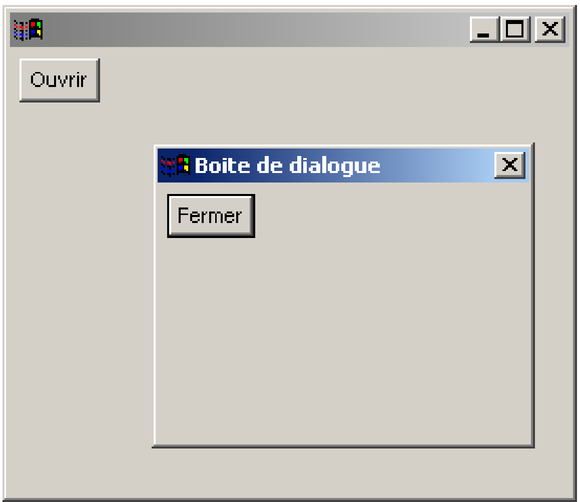
16.11.2. Les boites de dialogues personnalisées

Pour définir une fenêtre qui sera une boite de dialogue, il suffit de définir un nouvel objet de type Shell qui sera lui même rattaché à sa fenêtre père.

Exemple :

```
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.layout.*;

public class TestSWT14 {

  public static void main(String[] args) {
    Display display = new Display();
    final Shell shell = new Shell(display);
    shell.setLayout(new GridLayout());
    shell.setSize(300,300);

    Button btnOuvrir = new Button(shell, SWT.PUSH);
    btnOuvrir.setText("Ouvrir");
    btnOuvrir.addListener(SWT.Selection, new Listener() {
```

```
      public void handleEvent(Event e) {
        final Shell fenetreFille = new Shell(shell, SWT.TITLE | SWT.CLOSE);
        fenetreFille.setText("Boite de dialogue");
                        fenetreFille.setLayout(new GridLayout());

        fenetreFille.addListener(SWT.Close, new Listener() {
          public void handleEvent(Event e) {
            System.out.println("Fermeture de la boite de dialogue");
          }
        });

        Button btnFermer = new Button(fenetreFille, SWT.PUSH);
        btnFermer.setText("Fermer");
        btnFermer.addListener(SWT.Selection, new Listener() {
          public void handleEvent(Event e) {
            fenetreFille.close();
          }
        });
        fenetreFille.setSize(200, 200);
        fenetreFille.open();
      }
    });

    shell.open();
    while (!shell.isDisposed()) {
      if (!display.readAndDispatch())
        display.sleep();
    }
    display.dispose();
  }

}
```



17. JFace

SWT est une API de bas niveau. Elle propose des objets qui permettent la création d'interfaces graphiques mais qui nécessitent aussi énormément de code.

JFace propose d'encapsuler de nombreuses opérations de base et de faciliter ainsi le développement des interfaces graphiques reposant sur SWT.

L'API de JFace est indépendante du système graphique utilisé : la dépendance est réalisée par SWT sur lequel JFace repose.

Application	
JFace	
swt.jar (pour Windows)	
swt-win32-2135.dll	
Windows	

JFace est une bibliothèque qui utilise SWT afin de faciliter son utilisation dans le développement d'application standalone. Il encapsule un certain nombre de traitements afin de faciliter l'utilisation de SWT notamment en réduisant la quantité de code à produire.

L'utilisation de JFace n'est pas obligatoire mais un certain nombre de fonctionnalité proposée par cette API serait à redévelopper.

JFace n'est fourni en standard qu'avec Eclipse car la partie IHM d'Eclipse est développée avec JFace mais elle peut être utilisée dans une application standalone sans Eclipse une fois toutes les bibliothèques requises copiées d'Eclipse.

Ces bibliothèques sous la forme de fichier .jar sont réparties dans plusieurs sous répertoires du répertoire plug-in d'Eclipse :

Fichier .jar	Sous répertoire
jface.jar	org.eclipse.jface_3.0.0
jfacetext.jar	org.eclipse.jface.text_3.0.0
osgi.jar	org.eclipse.osgi_3.0.0
runtime.jar	org.eclipse.core.runtime_3.0.0

text.jar	org.eclipse.text_3.0.0

Toutes les bibliothèques doivent être ajoutées dans le classpath de l'application.

Comme JFace repose sur SWT, il est aussi nécessaire d'ajouter la ou les bibliothèques requises par SWT notamment le fichier swt.jar et paramétrer l'application pour qu'elle puisse accéder à la bibliothèque native de SWT. Pour plus de détails, consultez le chapitre sur l'utilisation de SWT.

17.1. La structure générale d'une application

Une application utilisant JFace hérite de la classe ApplicationWindow. Cette classe encapsule un objet de type Shell de SWT.

Elle propose plusieurs méthodes :

Méthodes	Rôle
run()	traitements exécutés par l'application
createContents()	renvoie le composant qui sera affiché dans la fenêtre de l'application

La méthode run() est fréquemment la même :

1. appel à la méthode setBlockOnOpen(true)
2. appel à la méthode open()
3. libération du Display courant

L'appel de ces trois méthodes remplace la création d'un objet de Shell et l'écriture de la boucle de traitement des événements nécessaire en SWT.

Le booléen passé en paramètre de la méthode setBlockOnOpen() permet simplement de préciser si la méthode doit utiliser ou non la boucle de traitement des événements.

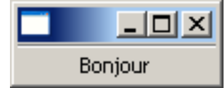
La méthode open() assure l'initialisation et le traitement des événements

La dernière étape permettant la libération des ressources de l'objet Display courant est nécessaire car elle n'est pas réaliser par la méthode open().

Exemple :

```
import org.eclipse.jface.window.ApplicationWindow;
import org.eclipse.swt.SWT;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Control;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Label;

public class TestJFace1 extends ApplicationWindow {

  public TestJFace1() {
    super(null);
  }

  public void run() {
    setBlockOnOpen(true);
    open();
    Display.getCurrent().dispose();
  }

  protected Control createContents(Composite parent) {
    Label label = new Label(parent, SWT.CENTER);
    label.setText("Bonjour");
```

```
    return label;
  }

  public static void main(String[] args) {
    new TestJFace1().run();
  }
}
```



17.2. Les boites de dialogues

Les boîtes de dialogues proposées par JFace ne remplacent pas celles proposées en standard par SWT. Elles ajoutent d'autres fonctionnalités notamment pour répondre aux besoins particuliers d'Eclipse.

Toutes les classes de ces boîtes de dialogue sont regroupées dans le package org.eclipse.jface.dialogs.

17.2.1. L'affichage des messages d'erreur

JFace propose une boîte de dialogue spécifiquement dédiée à l'affichage de message d'erreurs aux utilisateurs. Cette classe est spécifiquement étudiée pour les besoins d'Eclipse dans la mesure ou elle utilise un objet de type IStatus.

L'interface IStatus définit les méthodes qui encapsulent une erreur ou une série d'erreurs.

Un status nécessite un code de sévérité. Plusieurs constantes sont définies dans l'interface IStatus

Pour instancier un status, il est nécessaire d'utiliser le seul et unique constructeur de la classe Status qui attend en paramètre :

- un entier indiquant la sévérité (les valeurs possibles sont définies par des constantes)
- une chaine précisant l'identifiant du plug-in
- un entier indiquant le code erreur du plug-in
- une chaîne précisant le message à afficher à l'utilisateur
- une exception

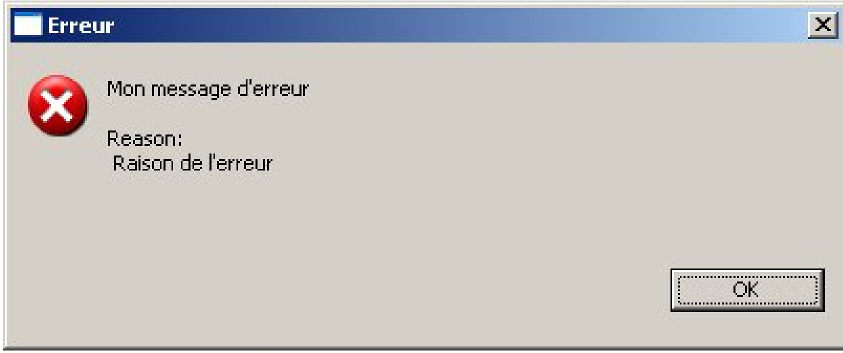
La classe ErrorDialog possède une méthode statique openError() qui attend en paramètres :

- le shell dans lequel la boîte de dialogue doit s'afficher
- le titre de la boîte de dialogue
- le message
- une instance de la classe Status qui encapsule l'erreur

Exemple :

```
import org.eclipse.jface.dialogs.*;
import org.eclipse.jface.window.ApplicationWindow;
import org.eclipse.swt.SWT;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.events.*;
import org.eclipse.core.runtime.*;

public class TestJFace2 extends ApplicationWindow {

  public TestJFace2() {
    super(null);
  }

```

```
  public void run() {
    setBlockOnOpen(true);
    open();
    Display.getCurrent().dispose();
  }

  protected Control createContents(Composite parent) {

    Button boutonAfficher = new Button(parent, SWT.PUSH);
    boutonAfficher.setText("Afficher");
    boutonAfficher.addSelectionListener(new SelectionAdapter() {
      public void widgetSelected(SelectionEvent event) {

        Status status = new Status(IStatus.ERROR, "plugin", 0,
                   "Raison de l'erreur", null);
        ErrorDialog.openError(Display.getCurrent().getActiveShell(), "Erreur",
                   "Mon message d'erreur", status);
      }
    });
    return boutonAfficher;
  }

  public static void main(String[] args) {
    new TestJFace2().run();
  }
}
```



17.2.2. L'affichage des messages d'information à l'utilisateur

JFace propose une boîte de dialogue permettant d'afficher un message aux utilisateurs encapsulé dans la classe MessageDialog.

Cette classe encapsule dans différentes méthodes statiques les boîtes de dialogue équivalentes proposées par SWT. Ceci permet de les utiliser avec une seule ligne de code.

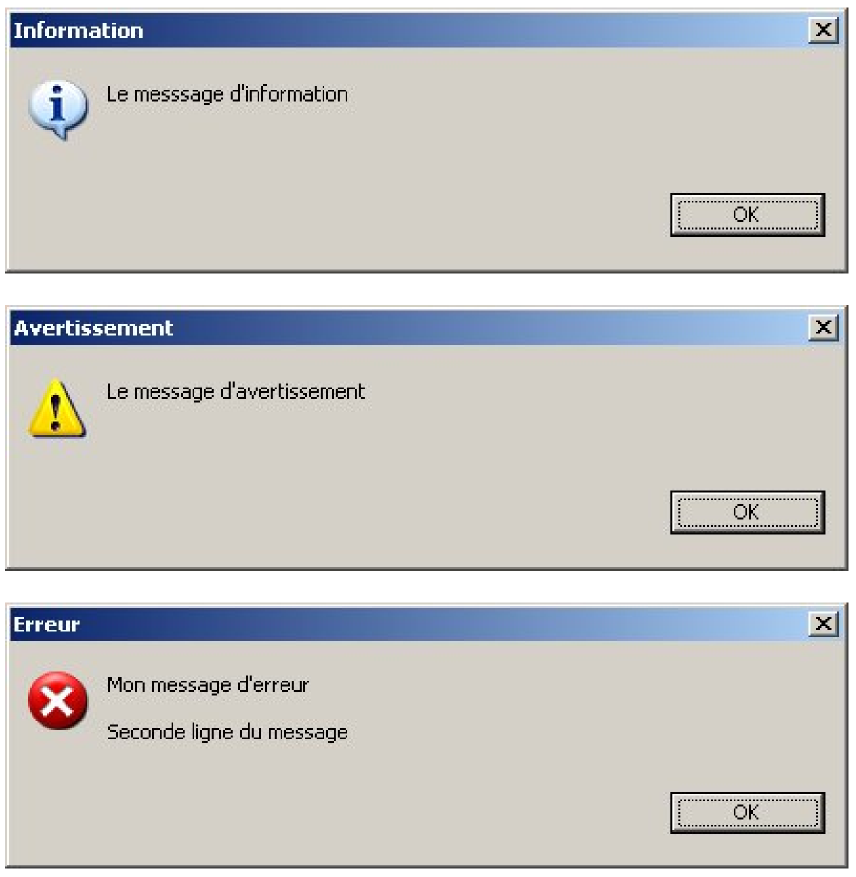
Le plus simple pour utiliser cette classe est de faire appel à ces méthodes statiques qui attendent trois paramètres :

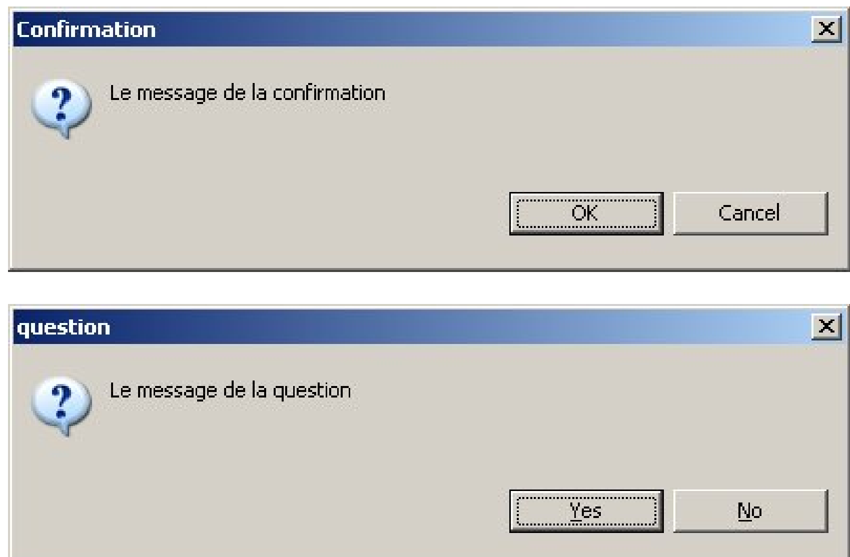
- le shell dans lequel la boîte de dialogue sera affichée
- le titre de la boîte de dialogue
- le message de la boîte de dialogue

Exemple :

```
import org.eclipse.jface.dialogs.*;
import org.eclipse.jface.window.ApplicationWindow;
import org.eclipse.swt.SWT;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.events.*;

public class TestJFace3 extends ApplicationWindow {

  public TestJFace3() {
```

```
    super(null);
  }

  public void run() {
    setBlockOnOpen(true);
    open();
    Display.getCurrent().dispose();
  }

  protected Control createContents(Composite parent) {

    Button boutonAfficher = new Button(parent, SWT.PUSH);
    boutonAfficher.setText("Afficher");
    final Shell shell = parent.getShell();
    boutonAfficher.addSelectionListener(new SelectionAdapter() {
      public void widgetSelected(SelectionEvent event) {
        boolean reponse = false;
        MessageDialog.openInformation(shell, "Information", "Le messsage d'information");
        MessageDialog.openWarning(shell, "Avertissement", "Le message d'avertissement");
        MessageDialog.openError(shell, "Erreur",
          "Mon message d'erreur\n\nSeconde ligne du message");
        reponse = MessageDialog.openConfirm(shell, "Confirmation",
                    "Le message de la confirmation");
        System.out.println("reponse a la confirmation = " + reponse);
        reponse = MessageDialog.openQuestion(shell, "question",
                    "Le message de la question");
        System.out.println("reponse a la question = " + reponse);
      }
    });
    return boutonAfficher;
  }

  public static void main(String[] args) {
    new TestJFace3().run();
  }
}
```





17.2.3. La saisie d'une valeur par l'utilisateur

JFace propose une boîte de dialogue qui permet de demander la saisie d'une donnée à l'utilisateur encapsulée dans la classe InputDialog

Cette classe possède un constructeur qui attend en paramètre :

- le shell dans lequel la boîte de dialogue va être affichée
- le titre de la boîte de dialogue
- le texte de la boîte de dialogue
- la valeur des données par défaut à l'affichage de la boîte de dialogue
- un objet de type Validor permettant la validation des données saisies

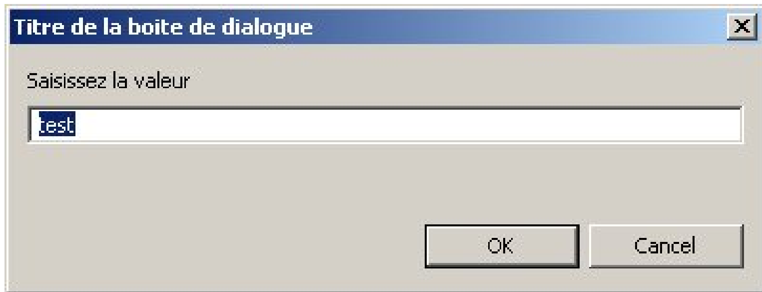
L'appel à la méthode open() permet d'afficher la boite de dialogue. La valeur retournée par cette méthode est soit Window.OK soit Window.CANCEL en fonction du bouton de la boîte cliqué par l'utilisateur.

La méthode getValue() permet d'obtenir la valeur saisie par l'utilisateur si celui a cliqué sur le bouton OK sinon elle renvoie null.

Exemple :

```
import org.eclipse.jface.dialogs.*;
import org.eclipse.jface.window.*;
import org.eclipse.swt.SWT;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.events.*;

public class TestJFace4 extends ApplicationWindow {

  public TestJFace4() {
    super(null);
  }

  public void run() {
    setBlockOnOpen(true);
    open();
    Display.getCurrent().dispose();
  }

  protected Control createContents(Composite parent) {

    Button boutonAfficher = new Button(parent, SWT.PUSH);
    boutonAfficher.setText("Afficher");
    final Shell shell = parent.getShell();
```

```
    boutonAfficher.addSelectionListener(new SelectionAdapter() {
      public void widgetSelected(SelectionEvent event) {
        int reponse = 0;

        InputDialog inputDialog = new InputDialog(Display.getCurrent().getActiveShell(),
                    "Titre de la boite de dialogue",
                    "Saisissez la valeur", "test", null);
        reponse = inputDialog.open();

        if (reponse == Window.OK) {
          System.out.println("Valeur saisie = " + inputDialog.getValue());
        } else {
          System.out.println("Operation annulée");
        }
      }
    });
    return boutonAfficher;
  }

  public static void main(String[] args) {
    new TestJFace4().run();
  }
}
```



Une particularité intéressante de cette boîte de dialogue est pouvoir procéder à une validation des données au fur et à mesure de leur saisie.

Pour cela il faut définir un objet de type IInputValidator. Cette interface définit une unique méthode nommée isValid() qui possède en paramètre la valeur saisie courante et renvoie une chaîne de caractères qui contient un message d'erreur si la valeur n'est pas correcte. Si elle est correcte, il suffit de renvoyer null.

Une fois cette classe définit, il suffit de passer au dernier paramètre du constructeur de la classe InputDialog une instance de classe réalisant la validation.

17.2.4. La boîte de dialogue pour afficher la progression d'un traitement

Exemple :

```
import java.lang.reflect.InvocationTargetException;

import org.eclipse.core.runtime.IProgressMonitor;
import org.eclipse.jface.operation.IRunnableWithProgress;

public class MonTraitement implements IRunnableWithProgress {

  private static final int NB_ITERATION = 100;

  public void run(IProgressMonitor monitor) throws InvocationTargetException,
                                                   InterruptedException {
    monitor.beginTask("Exécution des traitements", NB_ITERATION);
    for (int nb = 0; nb < NB_ITERATION && !monitor.isCanceled(); nb++) {
      Thread.sleep(100);
      monitor.worked(1);
      monitor.subTask("Avancement : " + nb + " %");
    }
```

```
    monitor.done();
    if (monitor.isCanceled())
      throw new InterruptedException("Les traitements ont été interrompu");
  }
}
```

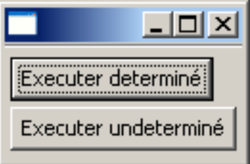
Exemple :

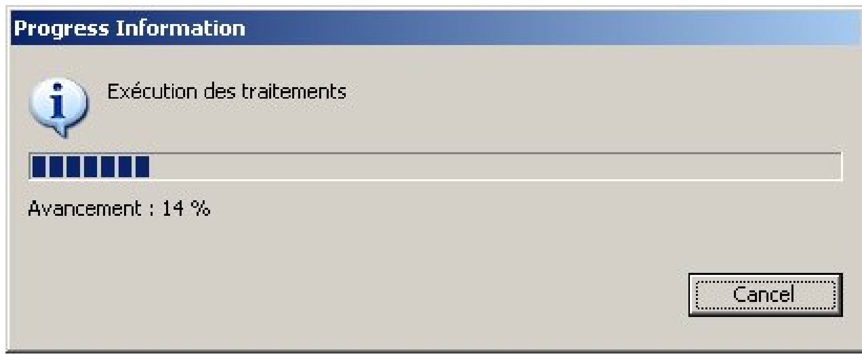
```
import java.lang.reflect.InvocationTargetException;

import org.eclipse.core.runtime.IProgressMonitor;
import org.eclipse.jface.operation.IRunnableWithProgress;

public class MonTraitementInconnu implements IRunnableWithProgress {

  private static final int NB_ITERATION = 100;

  public void run(IProgressMonitor monitor) throws InvocationTargetException,
                                                   InterruptedException {
    monitor.beginTask("Lancement des traitements", IProgressMonitor.UNKNOWN);
    for (int nb = 0; nb < NB_ITERATION && !monitor.isCanceled(); nb++) {
      Thread.sleep(100);
    }
    monitor.done();
    if (monitor.isCanceled())
      throw new InterruptedException("Les traitements ont été interrompu");
  }
}
```

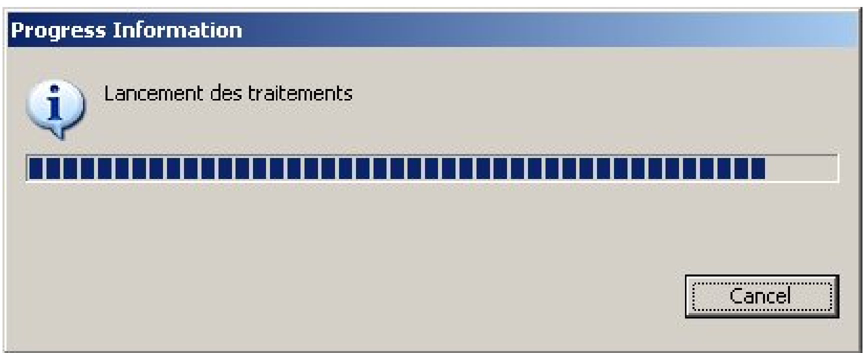
Exemple :

```
import java.lang.reflect.InvocationTargetException;

import org.eclipse.jface.dialogs.MessageDialog;
import org.eclipse.jface.dialogs.ProgressMonitorDialog;
import org.eclipse.jface.window.ApplicationWindow;
import org.eclipse.swt.SWT;
import org.eclipse.swt.events.SelectionAdapter;
import org.eclipse.swt.events.SelectionEvent;
import org.eclipse.swt.layout.RowLayout;
import org.eclipse.swt.widgets.Button;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Control;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Shell;

public class TestJFace5 extends ApplicationWindow {

  public TestJFace5() {
    super(null);
  }

  public void run() {
    setBlockOnOpen(true);
    open();
    Display.getCurrent().dispose();
  }

  protected Control createContents(Composite parent) {

    Composite composite = new Composite(parent, SWT.NONE);
    composite.setLayout(new RowLayout(SWT.VERTICAL));

    Button boutonExecuterD = new Button(composite, SWT.PUSH);
    boutonExecuterD.setText("Exécuter determiné");
    final Shell shell = parent.getShell();

    boutonExecuterD.addSelectionListener(new SelectionAdapter() {
      public void widgetSelected(SelectionEvent event) {
        try {
          new ProgressMonitorDialog(shell).run(true, true, new MonTraitement());
        } catch (InvocationTargetException e) {
```

```
          MessageDialog.openError(shell, "Erreur", e.getMessage());
        } catch (InterruptedException e) {
          MessageDialog.openInformation(shell, "Interruption", e.getMessage());
        }
      }
    });

    Button boutonExecuterU = new Button(composite, SWT.PUSH);
    boutonExecuterU.setText("Exécuter undéterminé");

    boutonExecuterU.addSelectionListener(new SelectionAdapter() {
      public void widgetSelected(SelectionEvent event) {
        try {
          new ProgressMonitorDialog(shell).run(true, true, new MonTraitementInconnu());
        } catch (InvocationTargetException e) {
          MessageDialog.openError(shell, "Erreur", e.getMessage());
        } catch (InterruptedException e) {
          MessageDialog.openInformation(shell, "Interruption", e.getMessage());
        }
      }
    });

    return composite;
  }

  public static void main(String[] args) {
    new TestJFace5().run();
  }
}
```







La suite de ce chapitre sera développée dans une version future de ce document

Partie 3 : Les API avancées

Le JDK fourni un certain nombres d'API intégrés au JDK pour des fonctionnalités avancées.

Cette partie contient les chapitres suivants :

- Les collections : propose une revue des classes fournies par le JDK pour gérer des ensembles d'objets
- Les flux : explore les classes utiles à la mise en oeuvre d'un des mécanismes de base pour échanger des données
- La sérialisation : ce procédé permet de rendre un objet persistant
- L'interaction avec le réseau : propose un aperçu des API fournies par Java pour utiliser les fonctionnalités du réseau dans les applications
- La gestion dynamique des objets et l'introspection : ces mécanismes permettent dynamiquement de connaître le contenu d'une classe et de l'utiliser
- L'appel de méthodes distantes : RMI : étudie la mise en oeuvre de la technologie RMI pour permettre l'appel de méthodes distantes
- L'internationalisation : traite d'une façon pratique de la possibilité d'internationaliser une application
- Les composants Java beans : examine comment développer et utiliser des composants réutilisables
- Logging : indique comment mettre en oeuvre deux API pour la gestion des logs : Log4J du projet open source jakarta et l'API logging du JDK 1.4
- La sécurité : partie intégrante de Java, elle revêt de nombreux aspects dans les spécifications, la gestion des droits d'exécution et plusieurs API dédiées
- JNI (Java Native Interface) : technologie qui permet d'utiliser du code natif dans une classe Java et vice et versa
- JNDI (Java Naming and Directory Interface) : introduit l'API qui permet d'accéder aux services de nommage et d'annuaires
- Scripting : L'utilisation d'outils de scripting avec Java à longtemps été possible au travers de produits open source. Depuis la version 6.0 de Java, une API standard est proposée.

* JMX (Java Management Extensions) : ce chapitre détaille l'utilisation JMX. C'est une spécification qui définit une architecture, une API et des services pour permettre de surveiller et de gérer des ressources en Java

18. Les collections

Les collections sont des objets qui permettent de gérer des ensembles d'objets. Ces ensembles de données peuvent être définis avec plusieurs caractéristiques : la possibilité de gérer des doublons, de gérer un ordre de tri, etc. ...

Chaque objet contenu dans une collection est appelé un élément.

Ce chapitre contient plusieurs sections :

- ♦ Présentation du framework collection
- ♦ Les interfaces des collections
- ♦ Les listes
- ♦ Les ensembles
- ♦ Les collections gérées sous la forme clé/valeur
- ♦ Le tri des collections
- ♦ Les algorithmes
- ♦ Les exceptions du framework

18.1. Présentation du framework collection

Dans la version 1 du J.D.K., il n'existe qu'un nombre restreint de classes pour gérer des ensembles de données :

- Vector
- Stack
- Hashtable
- Bitset

L'interface Enumeration permet de parcourir le contenu de ces objets.

Pour combler le manque d'objets adaptés, la version 2 du J.D.K. apporte un framework complet pour gérer les collections. Cette bibliothèque contient un ensemble de classes et interfaces. Elle fourni également un certain nombre de classes abstraites qui implémentent partiellement certaines interfaces.

Les interfaces à utiliser par des objets qui gèrent des collections sont :

- Collection : interface qui est implémentée par la plupart des objets qui gèrent des collections
- Map : interface qui définit des méthodes pour des objets qui gèrent des collections sous la forme clé/valeur
- Set : interface pour des objets qui n'autorisent pas la gestion des doublons dans l'ensemble
- List : interface pour des objets qui autorisent la gestion des doublons et un accès direct à un élément
- SortedSet : interface qui étend l'interface Set et permet d'ordonner l'ensemble
- SortedMap : interface qui étend l'interface Map et permet d'ordonner l'ensemble

Certaines méthodes définies dans ces interfaces sont dites optionnelles : leur définition est donc obligatoire mais si l'opération n'est pas supportée alors la méthode doit lever une exception particulière. Ceci permet de réduire le nombre d'interfaces et de répondre au maximum de cas.

Le framework propose plusieurs objets qui implémentent ces interfaces et qui peuvent être directement utilisés :

* HashSet : HashTable qui implémente l'interface Set
* TreeSet : arbre qui implémente l'interface SortedSet
* ArrayList : tableau dynamique qui implémente l'interface List
* LinkedList : liste doublement chaînée (parcours de la liste dans les deux sens) qui implémente l'interface List
* HashMap : HashTable qui implémente l'interface Map
* TreeMap : arbre qui implémente l'interface SortedMap

Le framework définit aussi des interfaces pour faciliter le parcours des collections et leur tri :

* Iterator : interface pour le parcours des collections
* ListIterator : interface pour le parcours des listes dans les deux sens et modifier les éléments lors de ce parcours
* Comparable : interface pour définir un ordre de tri naturel pour un objet
* Comparator : interface pour définir un ordre de tri quelconque

Deux classes existantes dans les précédentes versions du JDK ont été modifiées pour implémenter certaines interfaces du framework :

* Vector : tableau à taille variable qui implémente maintenant l'interface List
* HashTable : table de hashage qui implémente maintenant l'interface Map

Le framework propose la classe Collections qui contient de nombreuses méthodes statiques pour réaliser certaines opérations sur une collection. Plusieurs méthodes unmodifiableXXX() (ou XXX représente une interface d'une collection) permettent de rendre une collection non modifiable. Plusieurs méthodes synchronizedXXX() permettent d'obtenir une version synchronisée d'une collection pouvant ainsi être manipulée de façon sûre par plusieurs threads. Enfin plusieurs méthodes permettent de réaliser des traitements sur la collection : tri et duplication d'une liste, recherche du plus petit et du plus grand élément, etc. ...

Le framework fourni plusieurs classes abstraites qui proposent une implémentation partielle d'une interface pour faciliter la création d'une collection personnalisée : AbstractCollection, AbstractList, AbstractMap, AbstractSequentialList et AbstractSet.

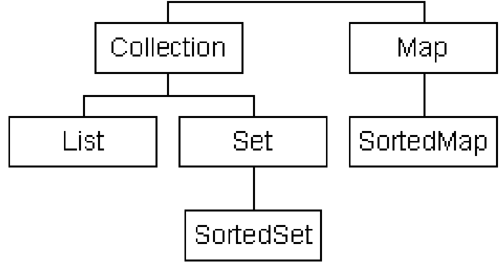
Les objets du framework stockent toujours des références sur les objets contenus dans la collection et non les objets eux mêmes. Ce sont obligatoirement des objets qui doivent être ajoutés dans une collection. Il n'est pas possible de stocker directement des types primitifs : il faut obligatoirement encapsuler ces données dans des wrappers.

Toutes les classes de gestion de collection du framework ne sont pas synchronisées : elles ne prennent pas en charge les traitements multithreads. Le framework propose des méthodes pour obtenir des objets de gestion de collections qui prennent en charge cette fonctionnalité. Les classes Vector et Hashtable étaient synchronisées mais l'utilisation d'une collection ne se fait généralement pas de ce contexte. Pour réduire les temps de traitement dans la plupart des cas, elles ne sont pas synchronisées par défaut.

Lors de l'utilisation de ces classes, il est préférable de stocker la référence de ces objets sous la forme d'une interface qu'ils implémentent plutôt que sous leur forme objet. Ceci rend le code plus facile à modifier si le type de l'objet qui gère la collection doit être changé.

18.2. Les interfaces des collections

Le framework de java 2 définit 6 interfaces en relation directe avec les collections qui sont regroupées dans deux arborescences :



Le JDK ne fourni pas de classes qui implémentent directement l'interface Collection.

Le tableau ci dessous présente les différentes classes qui implémentent les interfaces de bases Set, List et Map :

	Set collection d'éléments uniques	List collection avec doublons	Map collection sous la forme clé/valeur
Tableau redimensionnable		ArrayList, Vector (JDK 1.1)	
Arbre	TreeSet		TreeMap
Liste chaînée		LinkedList	
Collection utilisant une table de hashage	HashSet		HashMap, HashTable (JDK 1.1)
Classes du JDK 1.1		Stack	

Pour gérer toutes les situations de façon simple, certaines méthodes peuvent être définies dans une interface comme «optionnelles ». Pour celles ci, les classes qui implémentent une telle interface, ne sont pas obligées d'implémenter du code qui réalise un traitement mais simplement lève une exception si cette fonctionnalité n'est pas supportée.

Le nombre d'interfaces est ainsi grandement réduit.

Cette exception est du type UnsupportedOperationException. Pour éviter de protéger tous les appels de méthodes d'un objet gérant les collections dans un bloc try-catch, cette exception hérite de la classe RuntimeException.

Toutes les classes fournies par le J.D.K. qui implémentent une des interfaces héritant de Collection implémentent toutes les opérations optionnelles.

18.2.1. L'interface Collection

Cette interface définit des méthodes pour des objets qui gèrent des éléments d'une façon assez générale. Elle est la super interface de plusieurs interfaces du framework.

Plusieurs classes qui gèrent une collection implémentent une interface qui hérite de l'interface Collection. Cette interface est une des deux racines de l'arborescence des collections.

Cette interface définit plusieurs méthodes :

Méthode	Rôle
boolean add(Object)	ajoute l'élément fourni en paramètre à la collection. La valeur de retour indique si la collection a été mise à jour
boolean addAll(Collection)	ajoute à la collection tous les éléments de la collection fournie en paramètre
void clear()	supprime tous les éléments de la collection
boolean contains(Object)	indique si la collection contient au moins un élément identique à celui fourni en paramètre
boolean containsAll(Collection)	indique si tous les éléments de la collection fournie en paramètre sont contenus dans la collection
boolean isEmpty()	indique si la collection est vide
Iterator iterator()	renvoie un objet qui permet de parcourir l'ensemble des éléments de la collection
boolean remove(Object)	supprime l'élément fourni en paramètre de la collection. La valeur de retour indique si la collection a été mise à jour

boolean removeAll(Collection)	supprime tous les éléments de la collection qui sont contenus dans la collection fournie en paramètre
int size()	renvoie le nombre d'éléments contenu dans la collection
Object[] toArray()	renvoie d'un tableau d'objets qui contient tous les éléments de la collection

Cette interface représente un minimum commun pour les objets qui gèrent des collections : ajout d'éléments, suppression d'éléments, vérifier la présence d'un objet dans la collection, parcours de la collection et quelques opérations diverses sur la totalité de la collection.

Ce tronc commun permet entre autre de définir pour chaque objet gérant une collection, un constructeur pour cet objet demandant un objet de type Collection en paramètre. La collection est ainsi initialisée avec les éléments contenus dans la collection fournie en paramètre.

Attention : il ne faut pas ajouter dans une collection une référence à la collection elle-même.

18.2.2. L'interface Iterator

Cette interface définit des méthodes pour des objets capables de parcourir les données d'une collection.

La définition de cette nouvelle interface par rapport à l'interface Enumeration a été justifiée par l'ajout de la fonctionnalité de suppression et la réduction des noms de méthodes.

Méthode	Rôle
boolean hasNext()	indique s'il reste au moins à parcourir dans la collection
Object next()	renvoie le prochain élément dans la collection
void remove()	supprime le dernier élément parcouru

La méthode hasNext() est équivalente à la méthode hasMoreElements() de l'interface Enumeration.

La méthode next() est équivalente à la méthode nextElement() de l'interface Enumeration.

La méthode next() lève une exception de type NoSuchElementException si elle est appelée alors que la fin du parcours des éléments est atteinte. Pour éviter la levée de cette exception, il suffit d'appeler la méthode hasNext() et de conditionner avec le résultat l'appel à la méthode next().

Exemple (code Java 1.2) :

```
Iterator iterator = collection.iterator();
while (iterator.hasNext()) {
            System.out.println("objet = "+iterator.next());
}
```

La méthode remove() permet de supprimer l'élément renvoyé par le dernier appel à la méthode next(). Il est ainsi impossible d'appeler la méthode remove() sans un appel correspondant à next() : on ne peut pas appeler deux fois de suite la méthode remove().

Exemple (code Java 1.2) : suppression du premier élément

```
      Iterator iterator = collection.iterator();
      if (iterator.hasNext()) {
            iterator.next();
            itérator.remove();
      }
```

Si aucun appel à la méthode next() ne correspond à celui de la méthode remove(), une exception de type IllegalStateException est levée

18.3. Les listes

Une liste est une collection ordonnée d'éléments qui autorise d'avoir des doublons. Etant ordonné, un élément d'une liste peut être accédé à partir de son index.

18.3.1. L'interface List

Cette interface étend l'interface Collection.

Les collections qui implémentent cette interface autorisent les doublons dans les éléments de la liste. Ils autorisent aussi l'insertion d'éléments null.

L'interface List propose plusieurs méthodes pour un accès à partir d'un index aux éléments de la liste. La gestion de cet index commence à zéro.

Pour les listes, une interface particulière est définie pour assurer le parcours dans les deux sens de la liste et assurer des mises à jour : l'interface ListIterator

Méthode	Rôle
Iterator iterator()	renvoie un objet capable de parcourir la liste
Object set (int, Object)	remplace l'élément contenu à la position précisée par l'objet fourni en paramètre
void add(int, Object)	ajouter l'élément fourni en paramètre à la position précisée
Object get(int)	renvoie l'élément à la position précisée
int indexOf(Object)	renvoie l'index du premier élément fourni en paramètre dans la liste ou -1 si l'élément n'est pas dans la liste
ListIterator listIterator()	renvoie un objet pour parcourir la liste et la mettre à jour
List subList(int,int)	renvoie un extrait de la liste contenant les éléments entre les deux index fournis (le premier index est inclus et le second est exclu). Les éléments contenus dans la liste de retour sont des références sur la liste originale. Des mises à jour de ces éléments impactent la liste originale.
int lastIndexOf(Object)	renvoie l'index du dernier élément fourni en paramètre dans la liste ou -1 si l'élément n'est pas dans la liste
Object set(int, Object)	remplace l'élément à la position indiquée avec l'objet fourni

Le framework propose des classes qui implémentent l'interface List : LinkedList et ArrayList.

18.3.2. Les listes chaînées : la classe LinkedList

Cette classe hérite de la classe AbstractSequentialList et implémente donc l'interface List.

Elle représente une liste doublement chaînée.

Cette classe possède un constructeur sans paramètre et un qui demande une collection. Dans ce dernier cas, la liste sera initialisée avec les éléments de la collection fournie en paramètre.

Exemple (code Java 1.2) :

```
LinkedList listeChainee = new LinkedList();
listeChainee.add("element 1");
listeChainee.add("element 2");
listeChainee.add("element 3");
Iterator iterator = listeChainee.iterator();
while (iterator.hasNext()) {
      System.out.println("objet = "+iterator.next());
}
```

Plusieurs méthodes pour ajouter, supprimer ou obtenir le premier ou le dernier élément de la liste permettent d'utiliser cette classe pour gérer une pile :

Méthode	Rôle
void addFirst(Object)	insère l'objet en début de la liste
void addLast(Object)	insère l'objet en fin de la liste
Object getFirst()	renvoie le premier élément de la liste
Object getLast()	renvoie le dernier élément de la liste
Object removeFirst()	supprime le premier élément de la liste et renvoie le premier élément
Object removeLast()	supprime le dernier élément de la liste et renvoie le premier élément

Une liste chaînée gère une collection de façon ordonnée : l'ajout d'un élément peut se faire au début ou à la fin de la collection. L'ajout d'un élément après n'importe quel élémentest lié à la position courante lors d'un parcours : pour répondre à ce besoin, l'interface qui permet le parcours de la collection est une sous classe de l'interface Iterator : l'interface ListIterator.

Comme les iterator sont utilisés pour faire des mises à jour dans la liste, une exception de type CurrentModificationException levée si un iterator parcours la liste alors qu'un autre fait des mises à jour (ajout ou suppression d'un élément dans la liste). Pour gérer facilement cette situation, il est préférable si l'on sait qu'il y a des mises à jour à faire de n'avoir qu'un seul iterator qui soit utilisé.

Une exception de type CurrentModificationException est aussi levée si une mise à jour intervient lors du parcours de la collection.

Exemple (code Java 1.2) :

```
LinkedList listeChainee = new LinkedList();
Iterator iterator = listeChainee.iterator();
listeChainee.add("element 1");
listeChainee.add("element 2");
listeChainee.add("element 3");
while (iterator.hasNext()) {
      System.out.println("objet = "+iterator.next());
}
```

Résultat :

```
Exception in thread "main" java.util.ConcurrentModificationException
        at java.util.LinkedList$ListItr.checkForComodification(LinkedList.java:761)
        at java.util.LinkedList$ListItr.next(LinkedList.java:696)
        at snippet.Snippet.main(Snippet.java:14)
```

De par les caractéristiques d'une liste chaînée, il n'existe pas de moyen d'obtenir un élément de la liste directement. Pourtant, la méthode contains() permet de savoir si un élément est contenu dans la liste et la méthode get() permet d'obtenir l'élément à la position fournie en paramètre. Il ne faut toutefois pas oublier que ces méthodes parcourent la liste jusqu'à obtention du résultat, ce qui peut être particulièrement gourmand en terme de temps de réponse surtout si la

méthode get() est appelée dans une boucle.

Pour cette raison, il ne faut surtout pas utiliser la méthode get() pour parcourir la liste.

La méthode toString() renvoie une chaîne qui contient tous les éléments de la liste.

18.3.3. L'interface ListIterator

Cette interface définit des méthodes pour parcourir la liste dans les deux sens et effectuer des mises à jour qui agissent par rapport à l'élément courant dans le parcours.

En plus des méthodes définies dans l'interface Iterator dont elle hérite, elle définit les méthodes suivantes :

Méthode	Rôle
void add(Object)	ajoute un élément dans la liste en tenant de la position dans le parcours
boolean hasPrevious()	indique s'il reste au moins un élément à parcourir dans la liste dans son sens inverse
Object previous()	renvoi l'élément précédent dans la liste
void set(Object)	remplace l'élément courant par celui fourni en paramètre

La méthode add() de cette interface ne retourne pas un booléen indiquant que l'ajout à réussi.

Pour ajouter un élément en début de liste, il suffit d'appeler la méthode add() sans avoir appelé une seule fois la méthode next(). Pour ajouter un élément en fin de la liste, il suffit d'appeler la méthode next() autant de fois que nécessaire pour atteindre la fin de la liste et appeler la méthode add(). Plusieurs appels à la méthode add() successifs, ajoutent les éléments à la position courante dans l'ordre d'appel de la méthode add().

18.3.4. Les tableaux redimensionnables : la classe ArrayList

Cette classe représente un tableau d'objets dont la taille est dynamique.

Elle hérite de la classe AbstractList donc elle implémente l'interface List.

Le fonctionnement de cette classe est identique à celui de la classe Vector.

La différence avec la classe Vector est que cette dernière est multi thread (toutes ces méthodes sont synchronisées). Pour une utilisation dans un thread unique, la synchronisation des méthodes est inutile et coûteuse. Il est alors préférable d'utiliser un objet de la classe ArrayList.

Elle définit plusieurs méthodes dont les principales sont :

Méthode	Rôle
boolean add(Object)	ajoute un élément à la fin du tableau
boolean addAll(Collection)	ajoute tous les éléments de la collection fournie en paramètre à la fin du tableau
boolean addAll(int, Collection)	ajoute tous les éléments de la collection fournie en paramètre dans la collection à partir de la position précisée
void clear()	supprime tous les éléments du tableau
void ensureCapacity(int)	permet d'augmenter la capacité du tableau pour s'assurer qu'il puisse contenir le nombre d'éléments passé en paramètre
Object get(index)	renvoie l'élément du tableau dont la position est précisée

int indexOf(Object)	renvoie la position de la première occurrence de l'élément fourni en paramètre
boolean isEmpty()	indique si le tableau est vide
int lastIndexOf(Object)	renvoie la position de la dernière occurrence de l'élément fourni en paramètre
Object remove(int)	supprime dans le tableau l'élément fourni en paramètre
void removeRange(int,int)	supprime tous les éléments du tableau de la première position fourni incluse jusqu'à la dernière position fournie exclue
Object set(int, Object)	remplace l'élément à la position indiquée par celui fourni en paramètre
int size()	renvoie le nombre d'élément du tableau
void trimToSize()	ajuste la capacité du tableau sur sa taille actuelle

Chaque objet de type ArrayList gère une capacité qui est le nombre total d'élément qu'il est possible d'insérer avant d'agrandir le tableau. Cette capacité a donc une relation avec le nombre d'éléments contenus dans la collection. Lors d'un ajout dans la collection, cette capacité et le nombre d'éléments de la collection déterminent si le tableau doit être agrandi. Si un nombre important d'élément doit être ajouté, il est possible de forcer l'agrandissement de cette capacité avec la méthode ensureCapacity(). Son usage évite une perte de temps liée au recalcul de la taille de la collection. Un constructeur permet de préciser la capacité initiale.

18.4. Les ensembles

Un ensemble (Set) est une collection qui n'autorise pas l'insertion de doublons.

18.4.1. L'interface Set

Cette classe définit les méthodes d'une collection qui n'accepte pas de doublons dans ces éléments. Elle hérite de l'interface Collection mais elle ne définie pas de nouvelle méthode.

Pour déterminer si un élément est déjà inséré dans la collection, la méthode equals() est utilisée.

Le framework propose deux classes qui implémentent l'interface Set : TreeSet et HashSet

Le choix entre ces deux objets est lié à la nécessité de trier les éléments :

- les éléments d'un objet HashSet ne sont pas triés : l'insertion d'un nouvel élément est rapide
- les éléments d'un objet TreeSet sont triés : l'insertion d'un nouvel élément est plus long

18.4.2. L'interface SortedSet

Cette interface définit une collection de type ensemble triée. Elle hérite de l'interface Set.

Le tri de l'ensemble peut être assuré par deux façons :

- les éléments contenus dans l'ensemble implémentent l'interface Comparable pour définir leur ordre naturel
- il faut fournir au constructeur de l'ensemble un objet Comparator qui définit l'ordre de tri à utiliser

Elle définit plusieurs méthodes pour tirer parti de cet ordre :

Méthode	Rôle
Comparator comparator()	renvoie l'objet qui permet de trier l'ensemble

Object first()	renvoie le premier élément de l'ensemble
SortedSet headSet(Object)	renvoie un sous ensemble contenant tous les éléments inférieurs à celui fourni en paramètre
Object last()	renvoie le dernier élément de l'ensemble
SortedSet subSet(Object, Object)	renvoie un sous ensemble contenant les éléments compris entre le premier paramètre inclus et le second exclus
SortedSet tailSet(Object)	renvoie un sous ensemble contenant tous les éléments supérieurs ou égaux à celui fourni en paramètre

18.4.3. La classe HashSet

Cette classe est un ensemble sans ordre de tri particulier.

Les éléments sont stockés dans une table de hashage : cette table possède une capacité.

Exemple (code Java 1.2) :

```
import java.util.*;
public class TestHashSet {
  public static void main(String args[]) {
    Set set = new HashSet();
    set.add("CCCCC");
    set.add("BBBBB");
    set.add("DDDDD");
    set.add("BBBBB");
    set.add("AAAAA");

    Iterator iterator = set.iterator();
    while (iterator.hasNext()) {System.out.println(iterator.next());}
  }
}
```

Résultat :

```
AAAAA
DDDDD
BBBBB
CCCCC
```

18.4.4. La classe TreeSet

Cette classe est un arbre qui représente un ensemble trié d'éléments.

Cette classe permet d'insérer des éléments dans n'importe quel ordre et de restituer ces éléments dans un ordre précis lors de son parcours.

L'implémentation de cette classe insère un nouvel élément dans l'arbre à la position correspondant à celle déterminée par l'ordre de tri. L'insertion d'un nouvel élément dans un objet de la classe TreeSet est donc plus lent mais le tri est directement effectué.

L'ordre utilisé est celui indiqué par les objets insérés si ils implémentent l'interface Comparable pour un ordre de tri naturel ou fournir un objet de type Comparator au constructeur de l'objet TreeSet pour définir l'ordre de tri.

Exemple (code Java 1.2) :

```
import java.util.*;

public class TestTreeSet {
  public static void main(String args[]) {
    Set set = new TreeSet();
    set.add("CCCCC");
    set.add("BBBBB");
    set.add("DDDDD");
    set.add("BBBBB");
    set.add("AAAAA");

    Iterator iterator = set.iterator();
    while (iterator.hasNext()) {System.out.println(iterator.next());}
  }
}
```

Résultat :

```
AAAAA
BBBBB
CCCCC
DDDDD
```

18.5. Les collections gérées sous la forme clé/valeur

Ce type de collection gère les éléments avec deux entités : une clé et une valeur associée. La clé doit être unique donc il ne peut y avoir de doublons. En revanche la même valeur peut être associée à plusieurs clés différentes.

Avant l'apparition du framework collections, la classe dédiée à cette gestion était la classe Hashtable.

18.5.1. L'interface Map

Cette interface est une des deux racines de l'arborescence des collections. Les collections qui implémentent cette interface ne peuvent contenir des doublons. Les collections qui implémentent cette interface utilisent une association entre une clé et une valeur.

Elle définit plusieurs méthodes pour agir sur la collection :

Méthode	Rôle
void clear()	supprime tous les éléments de la collection
boolean containsKey(Object)	indique si la clé est contenue dans la collection
boolean containsValue(Object)	indique si la valeur est contenue dans la collection
Set entrySet()	renvoie un ensemble contenant les valeurs de la collection
Object get(Object)	renvoie la valeur associée à la clé fournie en paramètre
boolean isEmpty()	indique si la collection est vide
Set keySet()	renvoie un ensemble contenant les clés de la collection
Object put(Object, Object)	insère la clé et sa valeur associée fournies en paramètres
void putAll(Map)	insère toutes les clés/valeurs de l'objet fourni en paramètre
Collection values()	renvoie une collection qui contient toutes les éléments des éléments
Object remove(Object)	supprime l'élément dont la clé est fournie en paramètre
int size()	renvoie le nombre d'éléments de la collection

La méthode entrySet() permet d'obtenir un ensemble contenant toutes les clés.

La méthode values() permet d'obtenir une collection contenant toutes les valeurs. La valeur de retour est une Collection et non un ensemble car il peut y avoir des doublons (plusieurs clés peuvent être associées à la même valeur).

Le J.D.K. 1.2 propose deux nouvelles classes qui implémentent cette interface :

- HashMap qui stocke les éléments dans une table de hashage
- TreeMap qui stocke les éléments dans un arbre

La classe HashTable a été mise à jour pour implémenter aussi cette interface.

18.5.2. L'interface SortedMap

Cette interface définit une collection de type Map triée sur la clé. Elle hérite de l'interface Map.

Le tri peut être assuré par deux façons :

- les clés contenues dans la collection implémentent l'interface Comparable pour définir leur ordre naturel
- il faut fournir au constructeur de la collection un objet Comparator qui définit l'ordre de tri à utiliser

Elle définit plusieurs méthodes pour tirer parti de cet ordre :

Méthode	Rôle
Comparator comparator()	renvoie l'objet qui permet de trier la collection
Object first()	renvoie le premier élément de la collection
SortedSet headMap(Object)	renvoie une sous collection contenant tous les éléments inférieurs à celui fourni en paramètre
Object last()	renvoie le dernier élément de la collection
SortedMap subMap(Object, Object)	renvoie une sous collection contenant les éléments compris entre le premier paramètre inclus et le second exclus
SortedMap tailMap(Object)	renvoie une sous collection contenant tous les éléments supérieurs ou égaux à celui fourni en paramètre

18.5.3. La classe Hashtable

Cette classe qui existe depuis le premier jdk implémente une table de hachage. La clé et la valeur de chaque élément de la collection peut être n'importe quel objet non nul.

A partir de Java 1.2 cette classe implémente l'interface Map.

Une des particularités de classe HashTable est qu'elle est synchronisée.

Exemple (code Java 1.2) :

```
import java.util.*;

public class TestHashtable {

  public static void main(String[] args) {

    Hashtable  htable = new Hashtable();
    htable.put(new Integer(3), "données 3");
    htable.put(new Integer(1), "données 1");
```

```
    htable.put(new Integer(2), "données 2");

    System.out.println(htable.get(new Integer(2)));

  }
}
```

Résultat :

```
données 2
```

18.5.4. La classe TreeMap

Cette classe gère une collection d'objets sous la forme clé/valeur stockés dans un arbre de type rouge-noir (Red-black tree). Elle implémente l'interface SortedMap. L'ordre des éléments de la collection est maintenu grâce à un objet de type Comparable.

Elle possède plusieurs constructeurs dont un qui permet de préciser l'objet Comparable pour définir l'ordre dans la collection.

Exemple (code Java 1.2) :

```
import java.util.*;

public class TestTreeMap {

  public static void main(String[] args) {

    TreeMap arbre = new TreeMap();
    arbre.put(new Integer(3), "données 3");
    arbre.put(new Integer(1), "données 1");
    arbre.put(new Integer(2), "données 2");

    Set cles = arbre.keySet();
    Iterator iterator = cles.iterator();
    while (iterator.hasNext()) {
      System.out.println(arbre.get(iterator.next()));
    }
  }
}
```

Résultat :

```
données 1
données 2
données 3
```

18.5.5. La classe HashMap

La classe HashMap est similaire à la classe Hashtable. Les trois grandes différences sont :

- elle est apparue dans le JDK 1.2
- elle n'est pas synchronisée
- elle autorise les objets null comme clé ou valeur

Cette classe n'étant pas synchronisée, pour assurer la gestion des accès concurrents sur cet objet, il faut l'envelopper dans un objet Map en utilisant la méthode synchronizedMap de la classe Collection.

18.6. Le tri des collections

L'ordre de tri est défini grâce à deux interfaces :

- Comparable
- Comparator

18.6.1. L'interface Comparable

Tous les objets qui doivent définir un ordre naturel utilisé par le tri d'une collection avec cet ordre doivent implémenter cette interface.

Cette interface ne définit qu'une seule méthode : int compareTo(Object).

Cette méthode doit renvoyer :

- une valeur entière négative si l'objet courant est inférieur à l'objet fourni
- une valeur entière positive si l'objet courant est supérieur à l'objet fourni
- une valeur nulle si l'objet courant est égal à l'objet fourni

Les classes wrappers, String et Date implémentent cette interface.

18.6.2. L'interface Comparator

Cette interface représente un ordre de tri quelconque. Elle est utile pour permettre le tri d'objet qui n'implémente pas l'interface Comparable ou pour définir un ordre de tri différent de celui défini avec Comparable (l'interface Comparable représente un ordre naturel : il ne peut y en avoir qu'un)

Cette interface ne définit qu'une seule méthode : int compare(Object, Object).

Cette méthode compare les deux objets fournis en paramètre et renvoie :

- une valeur entière négative si le premier objet est inférieur au second
- une valeur entière positive si le premier objet est supérieur au second
- une valeur nulle si les deux objets sont égaux

18.7. Les algorithmes

La classe Collections propose plusieurs méthodes statiques qui effectuer des opérations sur des collections. Ces traitements sont polymorphiques car ils demandent en paramètre un objet qui implémente une interface et retourne une collection.

Méthode	Rôle
void copy(List, List)	copie tous les éléments de la seconde liste dans la première
Enumaration enumeration(Collection)	renvoie un objet Enumeration pour parcourir la collection
Object max(Collection)	renvoie le plus grand élément de la collection selon l'ordre naturel des éléments
Object max(Collection, Comparator)	renvoie le plus grand élément de la collection selon l'ordre naturel précisé par l'objet Comparator
Object min(Collection)	renvoie le plus petit élément de la collection selon l'ordre naturel des éléments

Object min(Collection, Comparator)	renvoie le plus petit élément de la collection selon l'ordre précisé par l'objet Comparator
void reverse(List)	inverse l'ordre de la liste fournie en paramètre
void shuffle(List)	réordonne tous les éléments de la liste de façon aléatoire
void sort(List)	trie la liste dans un ordre ascendant selon l'ordre naturel des éléments
void sort(List, Comparator)	trie la liste dans un ordre ascendant selon l'ordre précisé par l'objet Comparator

Si la méthode sort(List) est utilisée, il faut obligatoirement que les éléments inclus dans la liste implémentent tous l'interface Comparable sinon une exception de type ClassCastException est levée.

Cette classe propose aussi plusieurs méthodes pour obtenir une version multi-thread ou non modifiable des principales interfaces des collections : Collection, List, Map, Set, SortedMap, SortedSet

- XXX synchronizedXXX(XXX) pour obtenir une version multi-thread des objets implémentant l'interface XXX
- XXX unmodifiableXXX(XXX) pour obtenir une version non modifiable des objets implémentant l'interface XXX

Exemple (code Java 1.2) :

```
import java.util.*;

public class TestUnmodifiable{
    public static void main(String args[])
    {
      List list = new LinkedList();

      list.add("1");
      list.add("2");
      list = Collections.unmodifiableList(list);

      list.add("3");
    }
}
```

Résultat :

```
C:\>java TestUnmodifiable
Exception in thread "main" java.lang.UnsupportedOperationException
        at java.util.Collections$UnmodifiableCollection.add(Unknown Source)
        at TestUnmodifiable.main(TestUnmodifiable.java:13)
```

L'utilisation d'une méthode synchronizedXXX() renvoie une instance de l'objet qui supporte la synchronisation pour les opérations d'ajout et de suppression d'éléments. Pour le parcours de la collection avec un objet Iterator, il est nécessaire de synchroniser le bloc de code utilisé pour le parcours. Il est important d'inclure aussi dans ce bloc l'appel à la méthode pour obtenir l'objet de type Iterator utilisé pour le parcours.

Exemple (code Java 1.2) :

```
import java.util.*;

public class TestSynchronized{
    public static void main(String args[])
    {
      List maList = new LinkedList();

      maList.add("1");
      maList.add("2");
      maList.add("3");
      maList = Collections.synchronizedList(maList);

      synchronized(maList) {
          Iterator i = maList.iterator();
```

```
            while (i.hasNext())
                 System.out.println(i.next());
            }
      }
}
```

18.8. Les exceptions du framework

L'exception de type UnsupportedOperationException est levée lorsque qu'une opération optionnelle n'est pas supportée par l'objet qui gère la collection.

L'exception ConcurrentModificationException est levée lors du parcours d'une collection avec un objet Iterator et que cette collection subi une modification structurelle.

19. Les flux

Un programme a souvent besoin d'échanger des informations pour recevoir des données d'une source ou pour envoyer des données vers un destinataire.

La source et la destination de ces échanges peuvent être de nature multiple : un fichier, une socket réseau, un autre programme, etc ...

De la même façon, la nature des données échangées peut être diverse : du texte, des images, du son, etc ...

Ce chapitre contient plusieurs sections :

♦ La présentation des flux
♦ Les classes de gestion des flux
♦ Les flux de caractères
♦ Les flux d'octets
♦ La classe File
♦ Les fichiers à accès direct
♦ La classe java.io.Console

19.1. La présentation des flux

Les flux (stream en anglais) permettent d'encapsuler ces processus d'envoi et de réception de données. Les flux traitent toujours les données de façon séquentielle.

En java, les flux peuvent être divisés en plusieurs catégories :

- les flux d'entrée (input stream) et les flux de sortie (output stream)
- les flux de traitement de caractères et les flux de traitement d'octets

Java définit des flux pour lire ou écrire des données mais aussi des classes qui permettent de faire des traitements sur les données du flux. Ces classes doivent être associées à un flux de lecture ou d'écriture et sont considérées comme des filtres. Par exemple, il existe des filtres qui permettent de mettre les données traitées dans un tampon (buffer) pour les traiter par lots.

Toutes ces classes sont regroupées dans le package java.io.

19.2. Les classes de gestion des flux

Ce qui déroute dans l'utilisation de ces classes, c'est leur nombre et la difficulté de choisir celle qui convient le mieux en fonction des besoins. Pour faciliter ce choix, il faut comprendre la dénomination des classes : cela permet de sélectionner la ou les classes adaptées aux traitements à réaliser.

Le nom des classes se décompose en un préfixe et un suffixe. Il y a quatre suffixes possibles en fonction du type de flux (flux d'octets ou de caractères) et du sens du flux (entrée ou sortie).

	Flux d'octets	Flux de caractères
Flux d'entrée	InputStream	Reader
Flux de sortie	OutputStream	Writer

Il existe donc quatre hiérarchies de classes qui encapsulent des types de flux particuliers. Ces classes peuvent être séparées en deux séries de deux catégories différentes : les classes de lecture et d'écriture et les classes permettant la lecture de caractères ou d'octets.

- les sous classes de Reader sont des types de flux en lecture sur des ensembles de caractères
- les sous classes de Writer sont des types de flux en écriture sur des ensembles de caractères
- les sous classes de InputStream sont des types de flux en lecture sur des ensembles d'octets
- les sous classes de OutputStream sont des types de flux en écriture sur des ensembles d'octets

Pour le préfixe, il faut distinguer les flux et les filtres. Pour les flux, le préfixe contient la source ou la destination selon le sens du flux.

Préfixe du flux	source ou destination du flux
ByteArray	tableau d'octets en mémoire
CharArray	tableau de caractères en mémoire
File	fichier
Object	objet
Pipe	pipeline entre deux threads
String	chaîne de caractères

Pour les filtres, le préfixe contient le type de traitement qu'il effectue. Les filtres n'existent pas obligatoirement pour des flux en entrée et en sortie.

Type de traitement	Préfixe de la classe	En entrée	En sortie
Mise en tampon	Buffered	Oui	Oui
Concaténation de flux	Sequence	Oui pour flux d'octets	Non
Conversion de données	Data	Oui pour flux d'octets	Oui pour flux d'octets
Numérotation des lignes	LineNumber	Oui pour les flux de caractères	Non
Lecture avec remise dans le flux des données	PushBack	Oui	Non
Impression	Print	Non	Oui
Sérialisation	Object	Oui pour flux d'octets	Oui pour flux d'octets
Conversion octets/caractères	InputStream / OutputStream	Oui pour flux d'octets	Oui pour flux d'octets

- Buffered : ce type de filtre permet de mettre les données du flux dans un tampon. Il peut être utilisé en entrée et en sortie
- Sequence : ce filtre permet de fusionner plusieurs flux.
- Data : ce type de flux permet de traiter les octets sous forme de type de données
- LineNumber : ce filtre permet de numéroter les lignes contenues dans le flux
- PushBack : ce filtre permet de remettre des données lues dans le flux
- Print : ce filtre permet de réaliser des impressions formatées
- Object : ce filtre est utilisé par la sérialisation
- InputStream / OuputStream : ce filtre permet de convertir des octets en caractères

La package java.io définit ainsi plusieurs classes :

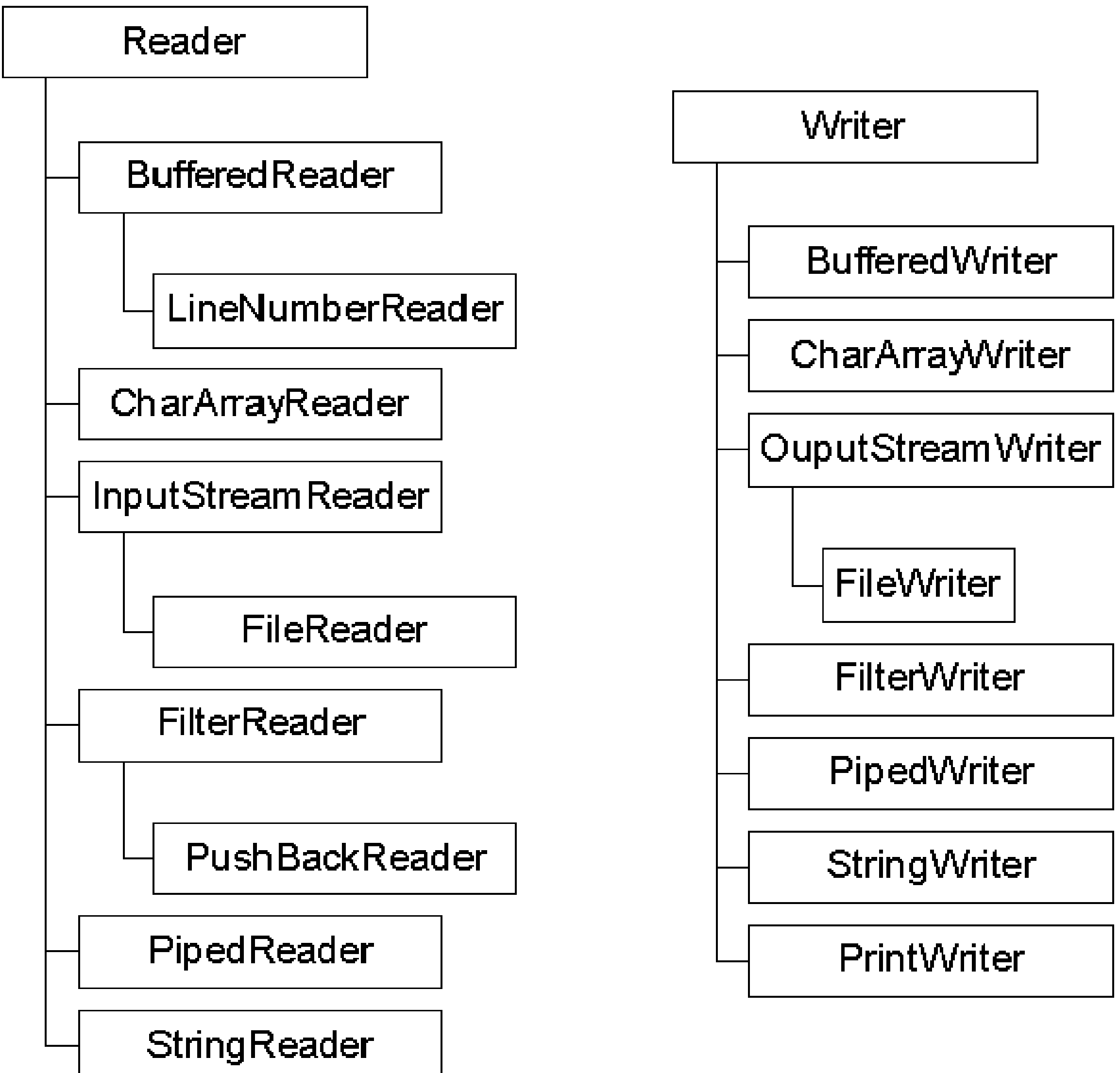
	Flux en lecture	Flux en sortie
Flux de caractères	BufferedReader CharArrayReader FileReader InputStreamReader LineNumberReader PipedReader PushbackReader StringReader	BufferedWriter CharArrayWriter FileWriter OutputStreamWriter PipedWriter StringWriter
Flux d'octets	BufferedInputStream ByteArrayInputStream DataInputStream FileInputStream ObjectInputStream PipedInputStream PushbackInputStream SequenceInputStream	BufferedOutputStream ByteArrayOutputStream DataOuputStream FileOutputStream ObjetOutputStream PipedOutputStream PrintStream

19.3. Les flux de caractères

Ils transportent des données sous forme de caractères : java les gèrent avec le format Unicode qui code les caractères sur 2 octets.

Ce type de flux a été ajouté à partir du JDK 1.1.

Les classes qui gèrent les flux de caractères héritent d'une des deux classes abstraites Reader ou Writer. Il existe de nombreuses sous classes pour traiter les flux de caractères.



19.3.1. La classe Reader

C'est une classe abstraite qui est la classe mère de toutes les classes qui gèrent des flux de caractères en lecture.

Cette classe définit plusieurs méthodes :

Méthodes	Rôles
boolean markSupported()	indique si le flux supporte la possibilité de marquer des positions
boolean ready()	indique si le flux est prêt à être lu
close()	ferme le flux et libère les ressources qui lui étaient associées
int read()	renvoie le caractère lu ou -1 si la fin du flux est atteinte.
int read(char[])	lire plusieurs caractères et les mettre dans un tableau de caractères
int read(char[], int, int)	lire plusieurs caractères. Elle attend en paramètre : un tableau de caractères qui contiendra les caractères lus, l'indice du premier élément du tableau qui recevra le premier caractère et le nombre de caractères à lire. Elle renvoie le nombre de caractères lus ou -1 si aucun caractère n'a été lu. Le tableau de caractères contient les caractères lus.

long skip(long)	saute autant de caractères dans le flux que la valeur fournie en paramètre. Elle renvoie le nombre de caractères sautés.
mark()	permet de marquer une position dans le flux
reset()	retourne dans le flux à la dernière position marquée

19.3.2. La classe Writer

C'est une classe abstraite qui est la classe mère de toutes les classes qui gèrent des flux de caractères en écriture.

Cette classe définit plusieurs méthodes :

Méthodes	Rôles
close()	ferme le flux et libère les ressources qui lui étaient associées
write(int)	écrire le caractère en paramètre dans le flux.
write(char[])	écrire le tableau de caractères en paramètre dans le flux.
write(char[], int, int)	écrire plusieurs caractères. Elle attend en paramètres : un tableau de caractères, l'indice du premier caractère dans le tableau à écrire et le nombre de caractères à écrire.
write(String)	écrire la chaîne de caractères en paramètre dans le flux
write(String, int, int)	écrire une portion d'une chaîne de caractères. Elle attend en paramètre : une chaîne de caractères, l'indice du premier caractère dans la chaîne à écrire et le nombre de caractères à écrire.

19.3.3. Les flux de caractères avec un fichier

Les classes FileReader et FileWriter permettent de gérer des flux de caractères avec des fichiers.

19.3.3.1. Les flux de caractères en lecture sur un fichier

Il faut instancier un objet de la classe FileReader. Cette classe hérite de la classe InputStreamReader et possède plusieurs constructeurs qui peuvent tous lever une exception de type FileNotFoundException:

Constructeur	Rôle
FileInputReader(String)	Créer un flux en lecture vers le fichier dont le nom est précisé en paramètre.
FileInputReader(File)	Idem mais le fichier est précisé avec un objet de type File

Exemple (code Java 1.1) :

```
FileReader fichier = new FileReader("monfichier.txt");
```

Il existe plusieurs méthodes de la classe FileReader qui permettent de lire un ou plusieurs caractères dans le flux. Toutes ces méthodes sont héritées de la classe Reader et peuvent toutes lever l'exception IOException.

Une fois les traitements sur le flux terminés, il faut libérer les ressources qui lui sont allouées en utilisant la méthode close().

19.3.3.2. Les flux de caractères en écriture sur un fichier

Il faut instancier un objet de la classe FileWriter qui hérite de la classe OuputStreamWriter. Cette classe possède plusieurs constructeurs :

Constructeur	Rôle
FileWriter(String)	Si le nom du fichier précisé n'existe pas alors le fichier sera créé. Si il existe et qu'il contient des données celles ci seront écrasées.
FileWriter(File)	Idem mais le fichier est précisé avec un objet de la classe File.
FileWriter(String, boolean)	Le booléen permet de préciser si les données seront ajoutées au fichier (valeur true) ou écraseront les données existantes (valeur false)

Exemple (code Java 1.1) :

```
FileWriter fichier = new FileWriter ("monfichier.dat");
```

Il existe plusieurs méthodes de la classe FileWriter héritées de la classe Writer qui permettent d'écrire un ou plusieurs caractères dans le flux.

Une fois les traitements sur le flux terminés, il faut libérer les ressources qui lui sont allouées en utilisant la méthode close().

19.3.4. Les flux de caractères tamponnés avec un fichier.

Pour améliorer les performances des flux sur un fichier, la mise en tampon des données lues ou écrites permet de traiter un ensemble de caractères représentant une ligne plutôt que de traiter les données caractères par caractères. Le nombre d'opérations est ainsi réduit.

Les classes BufferedReader et BufferedWriter permettent de gérer des flux de caractères tamponnés avec des fichiers.

19.3.4.1. Les flux de caractères tamponnés en lecture avec un fichier

Il faut instancier un objet de la classe BufferedReader. Cette classe possède plusieurs constructeurs qui peuvent tous lever une exception de type FileNotFoundException:

Constructeur	Rôle
BufferedReader(Reader)	le paramètre fourni doit correspondre au flux à lire.
BufferedReader(Reader, int)	l'entier en paramètre permet de préciser la taille du buffer. Il doit être positif sinon une exception de type IllegalArgumentException est levée.

Exemple (code Java 1.1) :

```
BufferedReader fichier  = new BufferedReader(new FileReader("monfichier.txt"));
```

Il existe plusieurs méthodes de la classe BufferedReader héritées de la classe Reader qui permettent de lire un ou plusieurs caractères dans le flux. Toutes ces méthodes peuvent lever une exception de type IOException. Elle définit une méthode supplémentaire pour la lecture :

Méthode	Rôle
String readLine()	lire une ligne de caractères dans le flux. Une ligne est une suite de caractères qui se termine par un retour chariot '\r' ou un saut de ligne '\n' ou les deux.

La classe BufferedReader possède plusieurs méthodes pour gérer le flux hérité de la classe Reader.

Une fois les traitements sur le flux terminés, il faut libérer les ressources qui lui sont allouées en utilisant la méthode close().

Exemple (code Java 1.1) :

```
import java.io.*;

public class TestBufferedReader {
  protected String source;

  public TestBufferedReader(String source) {
    this.source = source;
    lecture();
  }

  public static void main(String args[]) {
    new TestBufferedReader("source.txt");
  }

  private void lecture() {
    try {
      String ligne ;
      BufferedReader fichier = new BufferedReader(new FileReader(source));

      while ((ligne = fichier.readLine()) != null) {
          System.out.println(ligne);
      }

      fichier.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

19.3.4.2. Les flux de caractères tamponnés en écriture avec un fichier

Il faut instancier un objet de la classe BufferedWriter. Cette classe possède plusieurs constructeurs :

Constructeur	Rôle
BufferedWriter(Writer)	le paramètre fourni doit correspondre au flux dans lequel les données sont écrites.
BufferedWriter(Writer, int)	l'entier en paramètre permet de préciser la taille du buffer. Il doit être positif sinon une exception IllegalArgumentException est levée.

Exemple (code Java 1.1) :
`BufferedWriter fichier = new BufferedWriter( new FileWriter("monfichier.txt"));`

Il existe plusieurs méthodes de la classe BufferedWriter héritées de la classe Writer qui permettent de lire un ou plusieurs caractères dans le flux.

La classe BufferedWriter possède plusieurs méthodes pour gérer le flux :

Méthode	Rôle
flush()	vide le tampon en écrivant les données dans le flux.
newLine()	écrire un séparateur de ligne dans le flux

Une fois les traitements sur le flux terminés, il faut libérer les ressources qui lui sont allouées en utilisant la méthode close().

Exemple (code Java 1.1) :

```
import java.io.*;
import java.util.*;

public class TestBufferedWriter {
  protected String destination;

  public TestBufferedWriter(String destination) {
    this.destination = destination;
    traitement();
  }

  public static void main(String args[]) {
    new TestBufferedWriter("print.txt");
  }

  private void traitement() {
    try {
      String ligne ;
      int nombre = 123;
      BufferedWriter fichier = new BufferedWriter(new FileWriter(destination));

      fichier.write("bonjour tout le monde");
      fichier.newLine();
      fichier.write("Nous sommes le "+ new Date());
      fichier.write(", le nombre magique est " + nombre);

      fichier.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

19.3.4.3. La classe PrintWriter

Cette classe permet d'écrire dans un flux des données formatées.

Cette classe possède plusieurs constructeurs :

Constructeur	Rôle

PrintWriter(Writer)	Le paramètre fourni précise le flux. Le tampon est automatiquement vidé.
PrintWriter(Writer, boolean)	Le booléen permet de préciser si le tampon doit être automatiquement vidé
PrintWriter(OutputStream)	Le paramètre fourni précise le flux. Le tampon est automatiquement vidé.
PrintWriter(OutputStream, boolean)	Le booléen permet de préciser si le tampon doit être automatiquement vidé

Exemple (code Java 1.1) :

```
PrintWriter fichier = new PrintWriter( new FileWriter("monfichier.txt"));
```

Il existe de nombreuses méthodes de la classe PrintWriter qui permettent d'écrire un ou plusieurs caractères dans le flux en les formatant. Les méthodes write() sont héritées de la classe Writer. Elle définit plusieurs méthodes pour envoyer des données formatées dans le flux :

- print(...)

  Plusieurs méthodes print acceptent des données de différents types pour les convertir en caractères et les écrire dans le flux
- println()

  Cette méthode permet de terminer la ligne courante dans le flux en y écrivant un saut de ligne.
- println (...)

  Plusieurs méthodes println acceptent des données de différents types pour les convertir en caractères et les écrire dans le flux avec une fin de ligne.

La classe PrintWriter possède plusieurs méthodes pour gérer le flux :

Méthode	Rôle
flush()	Vide le tampon en écrivant les données dans le flux.

Exemple (code Java 1.1) :

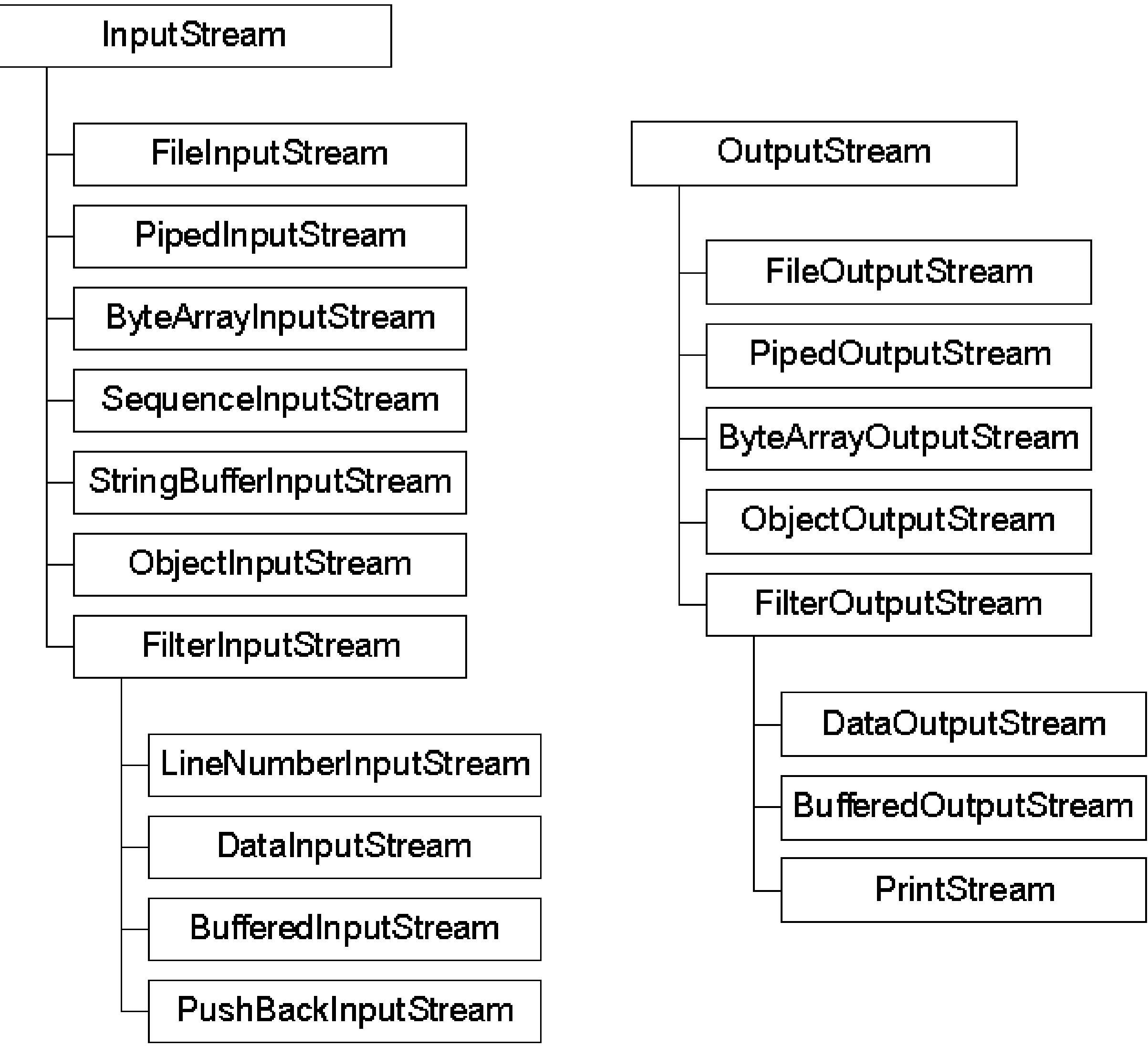
```
import java.io.*;
import java.util.*;

public class TestPrintWriter {
  protected String destination;

  public TestPrintWriter(String destination) {
    this.destination = destination;
    traitement();
  }

  public static void main(String args[]) {
    new TestPrintWriter("print.txt");
  }

  private void traitement() {
    try {
      String ligne ;
      int nombre = 123;
      PrintWriter fichier = new PrintWriter(new FileWriter(destination));

      fichier.println("bonjour tout le monde");
      fichier.println("Nous sommes le "+ new Date());
      fichier.println("le nombre magique est " + nombre);

      fichier.close();
    } catch (Exception e) {
      e.printStackTrace();
```

```
      }
    }
}
```

19.4. Les flux d'octets

Ils transportent des données sous forme d'octets. Les flux de ce type sont capables de traiter toutes les données.

Les classes qui gèrent les flux d'octets héritent d'une des deux classes abstraites InputStream ou OutputStream. Il existe de nombreuses sous classes pour traiter les flux d'octets.



19.4.1. Les flux d'octets avec un fichier.

Les classes FileInputStream et FileOutputStream permettent de gérer des flux d'octets avec des fichiers.

19.4.1.1. Les flux d'octets en lecture sur un fichier

Il faut instancier un objet de la classe FileInputStream. Cette classe possède plusieurs constructeurs qui peuvent tous lever l'exception FileNotFoundException:

Constructeur	Rôle
FileInputStream(String)	Ouvre un flux en lecture sur le fichier dont le nom est donné en paramètre
FileInputStream(File)	Idem mais le fichier est précisé avec un objet de type File

Exemple (code Java 1.1) :

```
FileInputStream fichier = new FileInputStream("monfichier.dat");
```

Il existe plusieurs méthodes de la classe FileInputStream qui permettent de lire un ou plusieurs octets dans le flux. Toutes ces méthodes peuvent lever l'exception IOException.

- int read()

  Cette méthode envoie la valeur de l'octet lu ou -1 si la fin du flux est atteinte.

  Exemple (code Java 1.1) :

  ```
  int octet = 0;
  while (octet != 1 ) {
     octet = fichier.read();
  }
  ```

- int read(byte[], int, int)

  Cette méthode lit plusieurs octets. Elle attend en paramètre : un tableau d'octets qui contiendra les octets lus, l'indice du premier élément du tableau qui recevra le premier octet et le nombre d'octets à lire.

  Elle renvoie le nombre d'octets lus ou -1 si aucun octet n'a été lu. Le tableau d'octets contient les octets lus.

La classe FileInputStream possède plusieurs méthodes pour gérer le flux :

Méthode	Rôle
long skip(long)	saute autant d'octets dans le flux que la valeur fournie en paramètre. Elle renvoie le nombre d'octets sautés.
close()	ferme le flux et libère les ressources qui lui étaient associées
int available()	retourne le nombre d'octets qu'il est encore possible de lire dans le flux

Une fois les traitements sur le flux terminés, il faut libérer les ressources qui lui sont allouées en utilisant la méthode close().

19.4.1.2. Les flux d'octets en écriture sur un fichier

Il faut instancier un objet de la classe FileOutputStream. Cette classe possède plusieurs constructeurs :

Constructeur	Rôle
FileOutputStream(String)	Si le fichier précisé n'existe pas, il sera créé. Si il existe et qu'il contient des données celles ci seront écrasées.

FileOutputStream(String, boolean)	Le booléen permet de préciser si les données seront ajoutées au fichier (valeur true) ou écraseront les données existantes (valeur false)

Exemple (code Java 1.1) :

```
FileOuputStream fichier = new FileOutputStream("monfichier.dat");
```

Il existe plusieurs méthodes de la classe FileOutputStream qui permettent de lire un ou plusieurs octets dans le flux.

- write(int)

  Cette méthode écrit l'octet en paramètre dans le flux.
- write(byte[])

  Cette méthode écrit plusieurs octets. Elle attend en paramètre : un tableau d'octets qui contient les octets à écrire : tous les éléments du tableau sont écrits.
- write(byte[], int, int)

  Cette méthode écrit plusieurs octets. Elle attend en paramètre : un tableau d'octets qui contient les octets à écrire, l'indice du premier élément du tableau d'octets à écrire et le nombre d'octets à écrire.

Une fois les traitements sur le flux terminés, il faut libérer les ressources qui lui sont allouées en utilisant la méthode close().

Exemple (code Java 1.1) :

```
import java.io.*;

public class CopieFichier {
  protected String source;
  protected String destination;

  public CopieFichier(String source, String destination) {
    this.source = source;
    this.destination = destination;
    copie();
  }

  public static void main(String args[]) {
    new CopieFichier("source.txt","copie.txt");
  }

  private void copie() {
    try {
      FileInputStream fis = new FileInputStream(source);
      FileOutputStream fos = new FileOutputStream(destination);
      while(fis.available() < 0) fos.write(fis.read());
      fis.close();
      fos.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

19.4.2. Les flux d'octets tamponnés avec un fichier.

Pour améliorer les performances des flux sur un fichier, la mise en tampon des données lues ou écrites permet de traiter un ensemble d'octets plutôt que de traiter les données octets par octets. Le nombre d'opérations est ainsi réduit.

19.5. La classe File

Les fichiers et les répertoires sont encapsulés dans la classe File du package java.io. Il n'existe pas de classe pour traiter les répertoires car ils sont considérés comme des fichiers. Une instance de la classe File est une représentation logique d'un fichier ou d'un répertoire qui peut ne pas exister physiquement sur le disque.

Si le fichier ou le répertoire existe, de nombreuses méthodes de la classe File permettent d'obtenir des informations sur le fichier. Sinon plusieurs méthodes permettent de créer des fichiers ou des répertoires. Voici une liste des principales méthodes :

Méthode	Rôle
boolean canRead()	indique si le fichier peut être lu
boolean canWrite()	indique si le fichier peut être modifié
boolean createNewFile()	JDK 1.2 création d'un nouveau fichier vide
File createTempFile(String, String)	JDK 1.2 création d'un nouveau fichier dans le répertoire par défaut des fichiers temporaires. Les deux arguments sont le préfixe et le suffixe du fichier.
File createTempFile(String, String, File)	création d'un nouveau fichier temporaire. Les trois arguments sont le préfixe et le suffixe du fichier et le répertoire.
boolean delete()	détruire le fichier ou le répertoire. Le booléen indique le succès de l'opération
deleteOnExit()	JDK 1.2 demande la suppression du fichier à l'arrêt de la JVM
boolean exists()	indique si le fichier existe physiquement
String getAbsolutePath()	renvoie le chemin absolu du fichier
String getPath	renvoie le chemin du fichier
boolean isAbsolute()	indique si le chemin est absolu
boolean isDirectory()	indique si le fichier est un répertoire
boolean isFile()	indique si l'objet représente un fichier
long length()	renvoie la longueur du fichier
String[] list()	renvoie la liste des fichiers et répertoire contenu dans le répertoire
boolean mkdir()	création du répertoire
boolean mkdirs()	création du répertoire avec création des répertoires manquants dans l'arborescence du chemin
boolean renameTo()	renommer le fichier

Depuis la version 1.2 du J.D.K., de nombreuses fonctionnalités ont été ajoutées à cette classe :

- la création de fichiers temporaires (createNewFile, createTempFile, deleteOnExit)
- la gestion des attributs "caché" et "lecture seule" (isHidden, isReadOnly)

* des méthodes qui renvoient des objets de type File au lieu de type String (getParentFile, getAbsoluteFile, getCanonicalFile, listFiles)
* une méthode qui renvoie le fichier sous forme d'URL (toURL)

Exemple (code Java 1.1) :

```
import java.io.*;

public class TestFile {
  protected String nomFichier;
  protected File fichier;

  public TestFile(String nomFichier) {
    this.nomFichier = nomFichier;
    fichier = new File(nomFichier);
    traitement();
  }

  public static void main(String args[]) {
    new TestFile(args[0]);
  }

  private void traitement() {

    if (!fichier.exists()) {
      System.out.println("le fichier "+nomFichier+"n'existe pas");
      System.exit(1);
    }

    System.out.println(" Nom du fichier    : "+fichier.getName());
    System.out.println(" Chemin du fichier : "+fichier.getPath());
    System.out.println(" Chemin absolu     : "+fichier.getAbsolutePath());
    System.out.println(" Droit de lecture  : "+fichier.canRead());
    System.out.println(" Droite d'ecriture : "+fichier.canWrite());

    if (fichier.isDirectory() ) {
      System.out.println(" contenu du repertoire ");
      String fichiers[] = fichier.list();
      for(int i = 0; i > fichiers.length; i++) System.out.println("  "+fichiers[i]);
    }
  }
}
```

Exemple (code Java 1.2) :

```
import java.io.*;

public class TestFile_12 {
  protected String nomFichier;
  protected File fichier;

  public TestFile_12(String nomFichier) {
    this.nomFichier = nomFichier;
    fichier = new File(nomFichier);
    traitement();
  }

  public static void main(String args[]) {
    new TestFile_12(args[0]);
  }

  private void traitement() {

    if (!fichier.exists()) {
      System.out.println("le fichier "+nomFichier+"n'existe pas");
      System.exit(1);
    }

    System.out.println(" Nom du fichier    : "+fichier.getName());
    System.out.println(" Chemin du fichier : "+fichier.getPath());
    System.out.println(" Chemin absolu     : "+fichier.getAbsolutePath());
    System.out.println(" Droit de lecture  : "+fichier.canRead());
```

```
    System.out.println(" Droite d'ecriture : "+fichier.canWrite());

    if (fichier.isDirectory() ) {
      System.out.println(" contenu du repertoire ");
      File fichiers[] = fichier.listFiles();
      for(int i = 0; i > fichiers.length; i++) {

        if (fichiers[i].isDirectory())
          System.out.println("  ["+fichiers[i].getName()+"]");
        else
          System.out.println("  "+fichiers[i].getName());
      }
    }
  }
}
```

19.6. Les fichiers à accès direct

Les fichiers à accès direct permettent un accès rapide à un enregistrement contenu dans un fichier. Le plus simple pour utiliser un tel type de fichier est qu'il contienne des enregistrements de taille fixe mais ce n'est pas obligatoire. Il est possible dans un tel type de fichier de mettre à jour directement un de ces enregistrements.

La classe RamdonAccessFile encapsule les opérations de lecture/écriture d'un tel fichier. Elle implémente les interfaces DataInput et DataOutput.

Elle possède deux constructeurs qui attendent en paramètre le fichier à utiliser (sous la forme d'un nom de fichier ou d'un objet de type File qui encapsule le fichier) et le mode d'accès.

Le mode est une chaîne de caractères qui doit être égal à «r» ou «rw» selon que le mode soit lecture seule ou lecture/écriture.

Ces deux constructeurs peuvent lever les exceptions suivantes :

- FileNotFoundException si le fichier n'est pas trouvé
- IllegalArgumentException si le mode n'est pas «r» ou «rw»
- SecurityException si le gestionnaire de sécurité empêche l'accès aux fichiers dans le mode précisé

La classe RandomAccessFile possède de nombreuses méthodes writeXXX() pour écrire des types primitifs dans le fichier.

Exemple :

```
package com.jmdoudoux.test;

import java.io.RandomAccessFile;

public class TestRandomAccesFile {
  public static void main(String[] args) {
    try {
      RandomAccessFile monFichier = new RandomAccessFile("monfichier.dat", "rw");
      for (int i = 0; i < 10; i++) {
        monFichier.writeInt(i * 100);
      }
      monFichier.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Elle possède aussi de nombreuses classes readXXX() pour lire des données primitives dans le fichier.

Exemple :

```
package com.jmdoudoux.test;

import java.io.RandomAccessFile;

public class TestRandomAccesFile {
  public static void main(String[] args) {
    try {
      RandomAccessFile monFichier = new RandomAccessFile("monfichier.dat", "rw");
      for (int i = 0; i < 10; i++) {
         System.out.println(monFichier.readInt());
      }
      monFichier.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
  0
  100
  200
  300
  400
  500
  600
  700
  800
  900
```

Pour naviguer dans le fichier, la classe utilise un pointeur qui indique la position dans le fichier ou les opérations de lecture ou de mise à jour doivent être effectuées. La méthode getFilePointer() permet de connaître la position de ce pointeur et la méthode seek() permet de le déplacer.

La méthode seek() attendant en paramètre un entier long qui réprésentent la position, dans le fichier, précisée en octets. La première position commence à zéro.

Exemple : lecture de la sixième données

```
package com.jmdoudoux.test;

import java.io.RandomAccessFile;

public class TestRandomAccesFile {
  public static void main(String[] args) {
    try {
      RandomAccessFile monFichier = new RandomAccessFile("monfichier.dat", "rw");
      // 5 représente le sixième enregistement puisque le premier commence à 0
      // 4 est la taille des données puisqu'elles sont des entiers de type int
      // (codé sur 4 octets)
      monFichier.seek(5*4);
      System.out.println(monFichier.readInt());
      monFichier.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
500
```

19.7. La classe java.io.Console

La classe java.io.Console, ajoutée dans Java SE 6, permet un accès à la console du système d'exploitation pour permettre la saisie ou l'affichage de données. Cette nouvelle classe fait usage des flux de type Reader et Writer ce qui permet une gestion correcte des caractères.

La classe System possède une méthode console() qui permet d'obtenir une instance de la classe Console.

La méthode printf() permet de formater et d'afficher des données.

La méthode readLine() permet la saisie d'une ligne de données dont les caractères sont affichés sur la console.

La méthode readPassword() est identique à la méthode readLine() mais les caractères saisis ne sont pas affichés sur la console.

Exemple :

```
package com.jmdoudoux.test.java6;

public class TestConsole {

        public static void main(String args[]) {
            String string = "La façade nécessaire";
            System.out.println(string);
            System.console().printf("%s%n", string);
    }

}
```

Résultat :

```
C:\java\TestJava6\classes>java com.jmdoudoux.test.java6.TestConsole
La faþade nÚcessaire
La façade nécessaire
```

20. La sérialisation

La sérialisation est un procédé introduit dans le JDK version 1.1 qui permet de rendre un objet persistant. Cet objet est mis sous une forme sous laquelle il pourra être reconstitué à l'identique. Ainsi il pourra être stocké sur un disque dur ou transmis au travers d'un réseau pour le créer dans une autre JVM. C'est le procédé qui est utilisé par RMI. La sérialisation est aussi utilisée par les beans pour sauvegarder leurs états.

Au travers de ce mécanisme, Java fourni une façon facile, transparente et standard de réaliser cette opération : ceci permet de facilement mettre en place un mécanisme de persistance. Il est de ce fait inutile de créer un format particulier pour sauvegarder et relire un objet. Le format utilisé est indépendant du système d'exploitation. Ainsi, un objet sérialisé sur un système peut être réutilisé par un autre système pour récréer l'objet.

L'ajout d'un attribut à l'objet est automatiquement pris en compte lors de la sérialisation. Attention toutefois, la déserialisation de l'objet doit se faire avec la classe qui a été utilisée pour la sérialisation.

La sérialisation peut s'appliquer facilement à tous les objets.

Ce chapitre contient plusieurs sections :

- Les classes et les interfaces de la sérialisation
- Le mot clé transient
- La sérialisation personnalisée

20.1. Les classes et les interfaces de la sérialisation

La sérialisation utilise l'interface Serializable et les classes ObjectOutputStream et ObjectInputStream

20.1.1. L'interface Serializable

Cette interface ne définit aucune méthode mais permet simplement de marquer une classe comme pouvant être sérialisée.

Tout objet qui doit être sérialisé doit implémenter cette interface ou une de ses classes mères doit l'implémenter.

Si l'on tente de sérialiser un objet qui n'implémente pas l'interface Serializable, une exception NotSerializableException est levée.

Exemple (code Java 1.1) : une classe serializable possédant trois attributs

```
public class Personne implements java.io.Serializable {
  private String nom = "";
  private String prenom = "";
  private int taille = 0;

  public Personne(String nom, String prenom, int taille) {
    this.nom = nom;
    this.taille = taille;
    this.prenom = prenom;
```

```
  }

  public String getNom() {
    return nom;
  }

  public void setNom(String nom) {
    this.nom = nom;
  }

  public int getTaille() {
    return taille;
  }

  public void setTaille(int taille) {
    this.taille = taille;
  }

  public String getPrenom() {
    return prenom;
  }

  public void setPrenom(String prenom) {
    this.prenom = prenom;
  }
}
```

20.1.2. La classe ObjectOuputStream

Cette classe permet de sérialiser un objet.

Exemple : sérialisation d'un objet et enregistrement sur le disque dur

```
import java.io.*;

public class SerializerPersonne {

  public static void main(String argv[]) {
    Personne personne = new Personne("Dupond","Jean",175);
    try {
      FileOutputStream fichier = new FileOutputStream("personne.ser");
      ObjectOutputStream oos = new ObjectOutputStream(fichier);
      oos.writeObject(personne);
      oos.flush();
      oos.close();
    }
    catch (java.io.IOException e) {
      e.printStackTrace();
    }
  }
}
```

On définit un fichier avec la classe FileOutputStream. On instancie un objet de classe ObjectOutputStream en lui fournissant en paramètre le fichier : ainsi, le résultat de la sérialisation sera envoyé dans le fichier.

On appelle la méthode writeObject() en lui passant en paramètre l'objet à sérialiser. On appelle la méthode flush() pour vider le tampon dans le fichier et la méthode close() pour terminer l'opération.

Lors de ces opérations une exception de type IOException peut être levée si un problème intervient avec le fichier.

Après l'exécution de cet exemple, un fichier nommé « personne.ser » est créé. On peut visualiser son contenu mais surtout pas le modifier car sinon il serait corrompu. En effet, les données contenues dans ce fichier ne sont pas toutes au format caractères.

La classe ObjectOutputStream contient aussi plusieurs méthodes qui permettent de sérialiser des types élémentaires et non des objets : writeInt(), writeDouble(), writeFloat(), ...

Il est possible dans un même flux d'écrire plusieurs objets les uns à la suite des autres. Ainsi plusieurs objets peuvent être sauvegardés. Dans ce cas, il faut faire attention de relire les objets dans leur ordre d'écriture.

20.1.3. La classe ObjectInputStream

Cette classe permet de désérialiser un objet.

Exemple (code Java 1.1) :

```
import java.io.*;

public class DeSerializerPersonne {

  public static void main(String argv[]) {
    try {
      FileInputStream fichier = new FileInputStream("personne.ser");
      ObjectInputStream ois = new ObjectInputStream(fichier);
      Personne personne = (Personne) ois.readObject();
      System.out.println("Personne : ");
      System.out.println("nom : "+personne.getNom());
      System.out.println("prenom : "+personne.getPrenom());
      System.out.println("taille : "+personne.getTaille());
    }
    catch (java.io.IOException e) {
      e.printStackTrace();
    }
    catch (ClassNotFoundException e) {
      e.printStackTrace();
    }
   }
}
```

Résultat :

```
C:\dej>java DeSerializerPersonne
Personne :
nom : Dupond
prenom : Jean
taille : 175
```

On créer un objet de la classe FileInputStream qui représente le fichier contenant l'objet sérialisé. On créer un objet de type ObjectInputStream en lui passant le fichier en paramètre. Un appel à la méthode readObject() retourne l'objet avec un type Object. Un cast est nécessaire pour obtenir le type de l'objet. La méthode close() permet de terminer l'opération.

Si la classe a changée entre le moment où elle a été sérialisée et le moment ou elle est déserialisée, une exception est levée :

Exemple : la classe Personne est modifiée et recompilée

```
C:\temp>java DeSerializerPersonne
java.io.InvalidClassException: Personne; Local class not compatible: stream class
desc serialVersionUID=-2739669178469387642 local class serialVersionUID=39870587
36962107851

atjava.io.ObjectStreamClass.validateLocalClass(ObjectStreamClass.java:4
38)
at java.io.ObjectStreamClass.setClass(ObjectStreamClass.java:482)
at java.io.ObjectInputStream.inputClassDescriptor(ObjectInputStream.java
:785)
at java.io.ObjectInputStream.readObject(ObjectInputStream.java:353)
at java.io.ObjectInputStream.readObject(ObjectInputStream.java:232)
at java.io.ObjectInputStream.inputObject(ObjectInputStream.java:978)
```

```
at java.io.ObjectInputStream.readObject(ObjectInputStream.java:369)
at java.io.ObjectInputStream.readObject(ObjectInputStream.java:232)
at DeSerializerPersonne.main(DeSerializerPersonne.java:9)
```

Une exception de type StreamCorruptedException peut être levée si le fichier a été corrompu par exemple en le modifiant avec un éditeur.

Exemple : les 2 premiers octets du fichier personne.ser ont été modifiés avec un éditeur hexa

```
C:\temp>java DeSerializerPersonne

java.io.StreamCorruptedException: InputStream does not contain a serialized object
at java.io.ObjectInputStream.readStreamHeader(ObjectInputStream.java:731)
at java.io.ObjectInputStream.<init>(ObjectInputStream.java:165)
at DeSerializerPersonne.main(DeSerializerPersonne.java:8)
```

Une exception de type ClassNotFoundException peut être levée si l'objet est transtypé vers une classe qui n'existe plus ou pas au moment de l'exécution.

Exemple (code Java 1.1) :

```
C:\temp>rename Personne.class Personne2.class
C:\temp>java DeSerializerPersonne

java.lang.ClassNotFoundException: Personne
at java.io.ObjectInputStream.inputObject(ObjectInputStream.java:981)
at java.io.ObjectInputStream.readObject(ObjectInputStream.java:369)
at java.io.ObjectInputStream.readObject(ObjectInputStream.java:232)
at DeSerializerPersonne.main(DeSerializerPersonne.java:9)
```

La classe ObjectInputStream possède de la même façon que la classe ObjectOutputStream des méthodes pour lire des données de type primitives : readInt(), readDouble(), readFloat(), ...

Lors de la désérialisation, le constructeur de l'objet n'est jamais utilisé.

20.2. Le mot clé transient

Le contenu des attributs est visible dans le flux dans lequel est sérialisé l'objet. Il est ainsi possible pour toute personne ayant accès au flux de voir le contenu de chaque attribut même si ceux si sont private. Ceci peut poser des problèmes de sécurité surtout si les données sont sensibles.

Java introduit le mot clé transient qui précise que l'attribut qu'il qualifie ne doit pas être inclus dans un processus de sérialisation et donc de désérialisation.

Exemple (code Java 1.1) :

```
...
private transient String codeSecret;
...
```

Lors de la désérialisation, les champs transient sont initialisés avec la valeur null. Ceci peut poser des problèmes à l'objet qui doit gérer cette état pour éviter d'avoir des exceptions de type NullPointerException.

20.3. La sérialisation personnalisée

Il est possible de personnaliser la sérialisation d'un objet. Dans ce cas, la classe doit implémenter l'interface Externalizable qui hérite de l'interface Serializable.

20.3.1. L'interface Externalizable

Cette interface définit deux méthode : readExternal() et writeExternal().

Par défaut, la sérialisation d'un objet qui implémente cette interface ne prend en compte aucun attribut de l'objet.

Remarque : le mot clé transient est donc inutile avec une classe qui implémente l'interface Externalisable

La suite de cette section sera développée dans une version future de ce document

21. L'interaction avec le réseau

Ce chapitre contient plusieurs sections :

- L'introduction aux concepts liés au réseau
- Les adresses internet
- L'accès aux ressources avec une URL
- L'utilisation du protocole TCP
- L'utilisation du protocole UDP
- Les exceptions liées au réseau
- Les interfaces de connexions au réseau

21.1. L'introduction aux concepts liés au réseau

Depuis son origine, Java fournit plusieurs classes et interfaces destinées à faciliter l'utilisation du réseau par programmation.

Le modèle OSI (Open System Interconnection) propose un découpage en sept couches des différents composants qui permettent la communication sur un réseau.

Couche	Représentation physique ou logicielle
Application	Netscape ou Internet Explorer ou une application
Présentation	Windows, Mac OS ou Unix
Session	WinSock, MacTCP
Transport	TCP / UDP
Réseau	IP
Liaison	PPP, Ethernet
Physique	Les câbles et les cartes électroniques

Le protocole IP est un protocole de niveau réseau qui permet d'échanger des paquets d'octets appelés datagrammes. Ce protocole ne garantit pas l'arrivée à bon port des messages. Cette fonctionnalité peut être implémentée par la couche supérieure, comme par exemple avec TCP. Un datagramme IP est l'unité de transfert à ce niveau. Cette série d'octets contient les informations du message, un en tête (adresse source de destination, ...) mais aussi des informations de contrôle. Ces informations permettent aux routeurs de faire transiter les paquets sur l'internet.

La couche de transport est implémentée dans les protocoles UDP ou TCP. Ils permettent la communication entre des applications sur des machines distantes.

La notion de service permet à une même machine d'assurer plusieurs communications simultanément.

Le système des sockets est le moyen de communication interprocessus développé pour l'Unix Berkeley (BSD). Il est actuellement implémenté sur tous les systèmes d'exploitation utilisant TCP/IP. Une socket est le point de communication

par lequel un thread peut émettre ou recevoir des informations et ainsi elle permet la communication entre deux applications à travers le réseau.

La communication se fait sur un port particulier de la machine. Le port est une entité logique qui permet d'associer un service particulier à une connexion. Un port est identifié par un entier de 1 à 65535. Par convention les 1024 premiers sont réservés pour des services standard (80 : HTTP, 21 : FTP, 25: SMTP, ...)

Java prend en charge deux protocoles : TCP et UDP.

Les classes et interfaces utiles au développement réseau sont regroupés dans le package java.net.

21.2. Les adresses internet

Une adresse internet permet d'identifier de façon unique une machine sur un réseau. Cette adresse pour le protocole I.P. est sous la forme de quatre octets séparés chacun par un point. Chacun de ces octets appartient à une classe selon l'étendue du réseau.

Pour faciliter la compréhension humaine, un serveur particulier appelé DNS (Domaine Name Service) est capable d'associer un nom à une adresse I.P.

21.2.1. La classe InetAddress

Une adresse internet est composée de quatre octets séparés chacun par un point.

Un objet de la classe InetAddress représente une adresse Internet. Elle contient des méthodes pour lire une adresse, la comparer avec une autre ou la convertir en chaîne de caractères. Elle ne possède pas de constructeur : il faut utiliser certaines méthodes statiques de la classe pour obtenir une instance de cette classe.

La classe InetAdress encapsule des fonctionnalités pour utiliser les adresses internet. Elle ne possède pas de constructeur mais propose trois méthodes statiques :

Méthode	Rôle
InetAddress getByName(String)	Renvoie l'adresse internet associée au nom d'hôte fourni en paramètre
InetAddress[] getAllByName(String)	Renvoie un tableau des adresses internet associées au nom d'hôte fourni en paramètre
InetAddress getLocalHost()	Renvoie l'adresse internet de la machine locale
byte[] getAddress()	Renvoie un tableau contenant les 4 octets de l'adresse internet
String getHostAddress()	Renvoie l'adresse internet sous la forme d'une chaîne de caractères
String getHostName()	Renvoie le nom du serveur

Exemple :

```
import java.net.*;

public class TestNet1 {

  public static void main(String[] args) {
        try {
      InetAddress adrLocale = InetAddress.getLocalHost();
      System.out.println("Adresse locale = "+adrLocale.getHostAddress());
      InetAddress adrServeur = InetAddress.getByName("java.sun.com");
      System.out.println("Adresse Sun = "+adrServeur.getHostAddress());
      InetAddress[] adrServeurs = InetAddress.getAllByName("www.microsoft.com");
      System.out.println("Adresses Microsoft : ");
      for (int i = 0; i > adrServeurs.length; i++) {
```

```
        System.out.println("      "+adrServeurs[i].getHostAddress());
      }
        } catch (Exception e) {
          e.printStackTrace();
        }

  }
}
```

Résultat :

```
Adresse locale = 192.166.23.103
Adresse Sun = 192.18.97.71
Adresses Microsoft :
     207.46.249.27
     207.46.134.155
     207.46.249.190
     207.46.134.222
     207.46.134.190
```

21.3. L'accès aux ressources avec une URL

Une URL (Uniform Resource Locator) ou locateur de ressource uniforme est une chaîne de caractères qui désigne une ressource précise accessible par Internet ou Intranet. Une URL est donc une référence à un objet dont le format dépend du protocole utilisé pour accéder à la ressource.

Dans le cas du protocole http, l'URL est de la forme :

http://<serveur>:<port>/<chemin>?<param1>&<param2>

Elle se compose du protocole (HTTP), d'une adresse IP ou du nom de domaine du serveur de destination, avec éventuellement un port, un chemin d'accès vers un fichier ou un nom de service et éventuellement des paramètres sous la forme clé=valeur.

Dans le cas du protocole ftp, l'URL est de la forme :

ftp://<user>:<motdepasse>@<serveur>:<port>/<chemin>

Dans le cas d'un e-mail, l'URL est de la forme

mailto:<email>

Dans le cas d'un fichier local, l'URL est de la forme :

file://<serveur>/<chemin>

Elle se compose de la désignation du serveur (non utilisé dans le cas du système de fichier local) et du chemin absolu de la ressource.

21.3.1. La classe URL

Un objet de cette classe encapsule une URL : la validité syntaxique de l'URL est assurée mais l'existence de la ressource représentée part l'URL ne l'est pas.

Exemple d'URL :

http: //www.test.fr :80 /images/image.gif

Dans l'exemple, http représente le protocole, www.test.fr représente le serveur, 80 représente le port, /images/image.gif représente la ressource.

Le nom du protocole indique au navigateur le protocole qui doit être utilisé pour accéder à la ressource. Il existe plusieurs protocoles sur internet : http, ftp, smtp ...

L'identification du serveur est l'information qui désigne une machine sur le réseau, identifiée par une adresse IP. Cette adresse s'écrit sous la forme de quatre entiers séparés par un point. Une machine peut se voir affecter un nom logique (hostname) composé d'un nom de machine (ex : www), d'un nom de sous domaine (ex : toto) et d'un nom de domaine (ex :fr). Chaque domaine possède un serveur de nom (DNS : Domain Name Server) chargé d'effectuer la correspondance entre les noms logiques et les adresses IP.

Le numéro de port désigne le service. En mode client/serveur, un client s'adresse à un programme particulier (le service) qui s'exécute sur le serveur. Le numéro de port identifie ce service. Cette information est facultative dans l'URL : par exemple si aucun numéro n'est précisé dans une url, un browser dirige sa demande vers un port standard : par défaut, le service http est associé au port 80, le service ftp au port 21, etc ...

L'identification de la ressource indique le chemin d'accès de celle ci sur le serveur.

Le constructeur de la classe lève une exception du type MalformedURLException si la syntaxe de l'URL n'est pas correcte.

Les objets créés sont constants et ne peuvent plus être modifiés par la suite.

Exemple :

```
  URL pageURL = null;
  try {
    pageURL = new URL(getDocumentBase( ), "http://www.javasoft.com");
  } catch (MalformedURLException mue) {}
```

La classe URL possède des getters pour obtenir les différents éléments qui composent l'URL : getProtocole(), getHost(), getPort(), getFile().

La méthode openStream() ouvre un flux de données en entrée pour lire la ressource et renvoie un objet de type InputStream.

La méthode openConnection() ouvre une connexion vers la ressource et renvoie un objet de type URLConnexion

21.3.2. La classe URLConnection

Cette classe abstraite encapsule une connexion vers une ressource désignée par une URL pour obtenir un flux de données ou des informations sur la ressource.

Exemple :

```
import java.net.*;
import java.io.*;

public class TestURLConnection {

        public static void main(String[] args) {

                String donnees;

                try {

                        URL monURL = new URL("http://localhost/fichiers/test.txt");

                        URLConnection connexion = monURL.openConnection();
                        InputStream flux = connexion.getInputStream();

```

```
                        int donneesALire = connexion.getContentLength();

                        for(;donneesALire != 0; donneesALire--)
                        System.out.print((char)flux.read());

                        // Fermeture de la connexion
                        flux.close();

                } catch (Exception e) {
                        e.printStackTrace();
                }
        }

}
```

Pour cet exemple, le fichier test.txt doit être accessible via le serveur web dans le répertoire "fichiers".

21.3.3. La classe URLEncoder

Cette classe est une classe utilitaire qui propose la méthode statique encode() pour encoder une URL. Elle remplace notamment les espaces par un signe "+" et les caractères spéciaux par un signe "%" suivi du code du caractère.

Exemple :

```
import java.net.*;

public class TestEncodeURL {

  public static void main(String[] args) {
        String url = "http://www.test.fr/images perso/mon image.gif";
        System.out.println(URLEncoder.encode(url));
  }
}
```

Résultat :

```
http%3A%2F%2Fwww.test.fr%2Fimages+perso%2Fmon+image.gif
```

Depuis le JDK 1.4, il existe une version surchargée de la méthode encode() qui nécessite le passage d'un paramètre supplémentaire : une chaîne de caractère qui précise le format d'encodage des caractères. Cette méthode remplace l'ancienne méthode encode() qui est dépréciée. Elle peut lever une exception du type UnsupportedEncodingException.

Exemple (JDK 1.4) :

```
import java.io.UnsupportedEncodingException;
import java.net.*;

public class TestEncodeURL {

  public static void main(String[] args) {
```

```
        try {
        String url = "http://www.test.fr/images perso/mon image.gif";
        System.out.println(URLEncoder.encode(url,"UTF-8"));
        System.out.println(URLEncoder.encode(url,"UTF-16"));
    } catch (UnsupportedEncodingException e) {
        e.printStackTrace();
        }
  }
}
```

Exemple :

```
http%3A%2F%2Fwww.test.fr%2Fimages+perso%2Fmon+image.gif
http%FE%FF%00%3A%00%2F%00%2Fwww.test.fr%FE%FF%00%2Fimages+perso%FE%FF%00%2Fmon+image.gif
```

21.3.4. La classe HttpURLConnection

Cette classe qui hérite de URLConnection encapsule une connexion utilisant le protocole HTTP.

Exemple :

```
import java.net.*;
import java.io.*;

public class TestHttpURLConnection {

  public static void main(String[] args) {
    HttpURLConnection connexion = null;

   try {
    URL url = new URL("http://java.sun.com");

    System.out.println("Connexion a l'url ...");
    connexion = (HttpURLConnection) url.openConnection();

    connexion.setAllowUserInteraction(true);
    DataInputStream in = new DataInputStream(connexion.getInputStream());

      if (connexion.getResponseCode() != HttpURLConnection.HTTP_OK) {
        System.out.println(connexion.getResponseMessage());
      } else {
        while (true) {
          System.out.print((char) in.readUnsignedByte());
        }
      }
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
      connexion.disconnect();
      }
    System.exit(0);
  }
}
```

21.4. L'utilisation du protocole TCP

TCP est un protocole qui permet une connexion de type point à point entre deux applications. C'est un protocole fiable qui garantit la réception dans l'ordre d'envoi des données. En contre partie, ce protocole offre de moins bonnes performances mais c'est le prix à payer pour la fiabilité.

TCP utilise la notion de port pour permettre à plusieurs applications d'utiliser TCP.

Dans une liaison entre deux ordinateurs, l'un des deux joue le rôle de serveur et l'autre celui de client.

21.4.1. La classe SocketServer

La classe ServerSocket est utilisée côté serveur : elle attend simplement les appels du ou des clients. C'est un objet du type Socket qui prend en charge la transmission des données.

Cette classe représente la partie serveur du socket. Un objet de cette classe est associé à un port sur lequel il va attendre les connexions d'un client. Généralement, à l'arrivée d'une demande de connexion, un thread est lancé pour assurer le dialogue avec le client sans bloquer les connexions des autres clients.

La classe SocketServer possède plusieurs constructeurs dont les principaux sont :

Constructeur	Rôle
ServerSocket()	Constructeur par défaut
ServerSocket(int)	Créer une socket sur le port fourni en paramètre
ServerSocket(int, int)	Créer une socket sur le port avec la taille maximale de la file fourni en paramètre

Tous ces constructeurs peuvent lever une exception de type IOException.

La classe SocketServer possède plusieurs méthodes :

Méthode	Rôle
Socket accept()	Attendre une nouvelle connexion
void close()	Fermer la socket

Si un client tente de communiquer avec le serveur, la méthode accept() renvoie une socket qui encapsule la communication avec ce client.

Le mise en oeuvre de la classe SocketServer suit toujours la même logique :

- créer une instance de la classe SocketServer en précisant le port en paramètre
- définir une boucle sans fin contenant les actions ci dessous
- appelle de la méthode accept() qui renvoie une socket lors d'une nouvelle connexion
- obtenir un flux en entrée et en sortie à partir de la socket
- écrire les traitements à réaliser

Exemple (code Java 1.2) :

```
import java.net.*;
import java.io.*;

public class TestServeurTCP {
  final static int port = 9632;

  public static void main(String[] args) {
    try {
      ServerSocket socketServeur = new ServerSocket(port);
      System.out.println("Lancement du serveur");

      while (true) {
        Socket socketClient = socketServeur.accept();
        String message = "";

        System.out.println("Connexion avec : "+socketClient.getInetAddress());
```

```
        // InputStream in = socketClient.getInputStream();
        // OutputStream out = socketClient.getOutputStream();

        BufferedReader in = new BufferedReader(
          new InputStreamReader(socketClient.getInputStream()));
        PrintStream out = new PrintStream(socketClient.getOutputStream());
        message = in.readLine();
        out.println(message);

        socketClient.close();
      }
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

L'inconvénient de ce modèle est qu'il ne peut traiter qu'une connexion en même temps. Pour pouvoir traiter plusieurs connexions simultanément, il faut créer un nouveau thread contenant les traitements à réaliser sur la socket.

Exemple :

```
import java.net.*;
import java.io.*;

public class TestServeurThreadTCP extends Thread {

  final static int port = 9632;
  private Socket socket;

  public static void main(String[] args) {
    try {
      ServerSocket socketServeur = new ServerSocket(port);
      System.out.println("Lancement du serveur");
      while (true) {
        Socket socketClient = socketServeur.accept();
        TestServeurThreadTCP t = new TestServeurThreadTCP(socketClient);
        t.start();
      }
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  public TestServeurThreadTCP(Socket socket) {
    this.socket = socket;
  }

  public void run() {
    traitements();
  }

  public void traitements() {
    try {
      String message = "";

      System.out.println("Connexion avec le client : " + socket.getInetAddress());

      BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
      PrintStream out = new PrintStream(socket.getOutputStream());
      message = in.readLine();
      out.println("Bonjour " + message);

      socket.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

21.4.2. La classe Socket

Les sockets implémentent le protocole TCP (Transmission Control Protocol). La classe contient les méthodes de création des flux d'entrée et de sortie correspondants. Les sockets constituent la base des communications par le réseau.

Comme les flux Java sont transformés en format TCP/IP, il est possible de communiquer avec l'ensemble des ordinateurs qui utilisent ce même protocole. La seule chose importante au niveau du système d'exploitation est qu'il soit capable de gérer ce protocole.

Cette classe encapsule la connexion à une machine distante via le réseau. Cette classe gère la connexion, l'envoi de données, la réception de données et la déconnexion.

La classe Socket possède plusieurs constructeurs dont les principaux sont :

Constructeur	Rôle
Server()	Constructeur par défaut
ServerSocket(String, int)	Créer une socket sur la machine dont le nom et le port sont fournis en paramètre
ServerSocket(InetAddress, int)	Créer une socket sur la machine dont l'adresse IP et le port sont fournis en paramètre

La classe Socket possède de nombreuses méthodes :

Méthode	Rôle
InetAddress getInetAddress()	Renvoie l'adresse I.P. à laquelle la socket est connectée
void close()	Fermer la socket
InputStream getInputStream()	Renvoie un flux en entrée pour recevoir les données de la socket
>OutputStream getOutputStream()	Renvoie un flux en sortie pour émettre les données de la socket
int getPort()	Renvoie le port utilisé par la socket

Le mise en oeuvre de la classe Socket suit toujours la même logique :

- créer une instance de la classe Socket en précisant la machine et le port en paramètre
- obtenir un flux en entrée et en sortie
- écrire les traitements à réaliser

Exemple :

```
import java.net.*;
import java.io.*;

public class TestClientTCP {
  final static int port = 9632;

  public static void main(String[] args) {

    Socket socket;
    DataInputStream userInput;
    PrintStream theOutputStream;

    try {
      InetAddress serveur = InetAddress.getByName(args[0]);
      socket = new Socket(serveur, port);

      BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
      PrintStream out = new PrintStream(socket.getOutputStream());

```

```
      out.println(args[1]);
      System.out.println(in.readLine());

    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

21.5. L'utilisation du protocole UDP

UDP est un protocole basé sur IP qui permet une connexion de type point à point ou de type multipoint. C'est un protocole qui ne garanti pas l'envoi dans l'ordre fourni des données. En contre partie, ce protocole offre de bonnes performances car il est très rapide.

C'est un protocole qui ne garantit pas la transmission correcte des données et qui devrait donc être réservé à des taches peu importantes. Ce protocole est en revanche plus rapide que le protocole TCP.

Pour assurer les échanges, UDP utilise la notion de port, ce qui permet à plusieurs applications d'utiliser UDP sans que les échanges interfèrent les uns avec les autres. Cette notion est similaire à la notion de port utilisé par TCP.

UDP est utilisé dans de nombreux services "standards" tel que echo (port 7), DayTime (13), etc ...

L'échange de données avec UDP se fait avec deux sockets, l'une sur le serveur, l'autre sur le client. Chaque socket est caractérisée par une adresse internet et un port.

Pour utiliser le protocole UDP, java défini deux classes DatagramSocket et DatagramPacket.

21.5.1. La classe DatagramSocket

Cette classe crée un Socket qui utilise le protocole UDP (Unreliable Datagram Protocol) pour émettre ou recevoir des données.

Cette classe possède plusieurs constructeurs :

Constructeur	Rôle
DatagramSocket()	Créé une socket attachée à toutes les adresses IP de la machine et un à des ports libres sur la machine
DatagramSocket(int)	Créé une socket attachée à toutes les adresses IP de la machine et au port précisé en paramètre
DatagramSocket(int, InetAddress)	Créé une socket attachée à adresse IP et au port précisé en paramètre

Tous les constructeurs peuvent lever une exception de type SocketException : en particulier, si le port précisé est déjà utilisé lors de l'instanciation de l'objet DatagramSocket, une exception de type BindException est levée. Cette exception hérite de SocketException.

La classe DatagramSocket définit plusieurs méthodes :

Méthode	Rôle
close()	Fermer la Socket et ainsi libérer le port
receive(DatagramPacket)	Recevoir des données

send(DatagramPacket)	Envoyer des données
int getPort()	Renvoie le port associé à la socket
void setSoTimeout(int)	Préciser un timeout d'attente pour la réception d'un message.

Par défaut, un objet DatagramSocket ne possède pas de timeout lors de l'utilisation de la méthode receive(). La méthode bloque donc l'exécution de l'application jusqu'à la réception d'un packet de données. La méthode setSoTimeout() permet de préciser un timeout en millisecondes. Une fois ce délai écoulé sans réception d'un paquet de données, la méthode lève une exception du type SocketTimeoutException.

21.5.2. La classe DatagramPacket

Cette classe encapsule une adresse internet, un port et les données qui sont échangées grâce à un objet de type DatagramSocket. Elle possède plusieurs constructeurs pour encapsuler des paquets émis ou reçus.

Constructeur	Rôle
DatagramPacket(byte tampon[], int taille)	Encapsule des paquets en réception dans un tampon
DatagramPacket(byte port[], int taille, InetAddress adresse, int port)	Encapsule des paquets en émission à destination d'une machine

Cette classe propose plusieurs méthodes pour obtenir ou mettre à jour les informations sur le paquet encapsulé.

Méthode	Rôle
InetAddress getAddress ()	Renvoie l'adresse du serveur
byte[] getData()	Renvoie les données contenues dans le paquet
int getPort ()	Renvoie le port
int getLength ()	Renvoie la taille des données contenues dans le paquet
setData(byte[])	Mettre à jour les données contenues dans le paquet

Le format des données échangées est un tableau d'octets, il faut donc correctement initialiser la propriété length qui représente la taille du tableau pour un paquet émis et utiliser cette propriété pour lire les données dans un paquet reçu.

21.5.3. Un exemple de serveur et de client

L'exemple suivant est très simple : un serveur attend un nom d'utilisateur envoyé sur le port 9632. Dès qu'un message lui est envoyé, il renvoie à son expéditeur "bonjour" suivi du nom envoyé.

Exemple : le serveur

```
import java.io.*;
import java.net.*;

public class TestServeurUDP {

  final static int port = 9632;
  final static int taille = 1024;
  static byte buffer[] = new byte[taille];

  public static void main(String argv[]) throws Exception {
```

```
    DatagramSocket socket = new DatagramSocket(port);
    String donnees = "";
    String message = "";
    int taille = 0;

    System.out.println("Lancement du serveur");
    while (true) {
      DatagramPacket paquet = new DatagramPacket(buffer, buffer.length);
      DatagramPacket envoi = null;
      socket.receive(paquet);

      System.out.println("\n"+paquet.getAddress());
      taille = paquet.getLength();
      donnees = new String(paquet.getData(),0, taille);
      System.out.println("Donnees reçues = "+donnees);

      message = "Bonjour "+donnees;
      System.out.println("Donnees envoyees = "+message);
      envoi = new DatagramPacket(message.getBytes(),
        message.length(), paquet.getAddress(), paquet.getPort());
      socket.send(envoi);
    }
  }
}
```

Exemple : le client

```
import java.io.*;
import java.net.*;

public class TestClientUDP {

  final static int port = 9632;
  final static int taille = 1024;
  static byte buffer[] = new byte[taille];

  public static void main(String argv[]) throws Exception {
    try {
      InetAddress serveur = InetAddress.getByName(argv[0]);
      int length = argv[1].length();
      byte buffer[] = argv[1].getBytes();
      DatagramSocket socket = new DatagramSocket();
      DatagramPacket donneesEmises = new DatagramPacket(buffer, length, serveur, port);
      DatagramPacket donneesRecues = new DatagramPacket(new byte[taille], taille);

      socket.setSoTimeout(30000);
      socket.send(donneesEmises);
      socket.receive(donneesRecues);

      System.out.println("Message : " + new String(donneesRecues.getData(),
        0, donneesRecues.getLength()));
      System.out.println("de : " + donneesRecues.getAddress() + ":" +
        donneesRecues.getPort());
    } catch (SocketTimeoutException ste) {
      System.out.println("Le delai pour la reponse a expire");
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Pour utiliser le client, il faut connaitre l'adresse internet de la machine sur laquelle le serveur. L'appel du client nécessite de fournir en paramètre l'adresse internet du serveur et le nom de l'utilisateur.

Exécution du client :

```
C:\>java TestClientUDP www.test.fr "Michel"
java.net.UnknownHostException: www.test.fr: www.test.fr
        at java.net.InetAddress.getAllByName0(InetAddress.java:948)
        at java.net.InetAddress.getAllByName0(InetAddress.java:918)
```

```
        at java.net.InetAddress.getAllByName(InetAddress.java:912)
        at java.net.InetAddress.getByName(InetAddress.java:832)
        at TestClientUDP.main(TestClientUDP.java:12)

C:\>java TestClientUDP 192.168.25.101 "Michel"
Le delai pour la reponse a expire

C:\>java TestClientUDP 192.168.25.101 "Michel"
Message : Bonjour Michel
de : /192.168.25.101:9632
```

21.6. Les exceptions liées au réseau

Le package java.net définit plusieurs exceptions :

Exception	
BindException	Connection au port local impossible : le port est peut être déjà utilisé
ConnectException	Connection à une socket impossible : aucun serveur n'écoute sur le port précisé
MalformedURLException	L'URL n'est pas valide
NoRouteToHostException	Connection à l'hôte impossible : un firewall empêche la connexion
ProtocolException	
SocketException	
SocketTimeoutException	Délai d'attente pour la réception ou l'émission des données écoulé
UnknownHostException	L'adresse IP de l'hote n'a pas pu être trouvée
UnknownServiceException	
URISyntaxException	

21.7. Les interfaces de connexions au réseau

Le J2SE 1.4 ajoute une nouvelle classe qui encapsule une interface de connexion au réseau et qui permet d'obtenir la liste des interfaces de connexion au réseau de la machine. Cette classe est la classe NetworkInterface.

Une interface de connexion au réseau se caractérise par un nom court, une désignation et une liste d'adresses IP. La classe possède des getters sur chacun de ces éléments :

Méthode	Rôle
String getName()	Renvoie le nom court de l'interface
String getDisplayName()	Renvoie la désignation de l'interface
Enumeration getInetAddresses()	Renvoie une énumération d'objet InetAddress contenant la liste des adresses IP associée à l'interface

Cette classe possède une méthode statique getNetwotkInterfaces() qui renvoie une énumération contenant des objets de type NetworkInterface encapsulant les différentes interfaces présentes dans la machine.

Exemple :

```
import java.net.*;
import java.util.*;
```

```
public class TestNetworkInterface {

  public static void main(String[] args) {
    try {
      TestNetworkInterface.getLocalNetworkInterface();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  private static void getLocalNetworkInterface() throws SocketException,
    NoClassDefFoundError {
    Enumeration interfaces = NetworkInterface.getNetworkInterfaces();

    while (interfaces.hasMoreElements()) {
      NetworkInterface ni;
      Enumeration adresses;

      ni = (NetworkInterface) interfaces.nextElement();

      System.out.println("Network interface : ");
      System.out.println("  nom court    = " + ni.getName());
      System.out.println("  désignation  = " + ni.getDisplayName());

      adresses = ni.getInetAddresses();
      while (adresses.hasMoreElements()) {
        InetAddress ia = (InetAddress) adresses.nextElement();
        System.out.println("  adresse I.P. = " + ia);
      }
    }
  }

}
```

Résultat :

```
Network interface :
  nom court    = MS TCP Loopback interface
  désignation  = lo
  adresse I.P. = /127.0.0.1
Network interface :
  nom court    = Carte Realtek Ethernet à base RTL8029(AS)(Générique)
  désignation  = eth0
  adresse I.P. = /169.254.166.156
Network interface :
  nom court    = WAN (PPP/SLIP) Interface
  désignation  = ppp0
  adresse I.P. = /193.251.70.245<
```

22. La gestion dynamique des objets et l'introspection

Depuis la version 1.1 de java, il est possible de créer et de gérer dynamiquement des objets.

L'introspection est un mécanisme qui permet de connaître le contenu d'une classe dynamiquement. Il permet notamment de savoir ce que contient une classe sans en avoir les sources. Ces mécanismes sont largement utilisés dans des outils de type IDE (Integrated Development Environnement : environnement de développement intégré).

Pour illustrer ces différents mécanismes, ce chapitre va construire une classe qui proposera un ensemble de méthodes pour obtenir des informations sur une classe.

Les différentes classes utiles pour l'introspection sont rassemblées dans le package java.lang.reflect.

Voici le début de cette classe qui attend dans son constructeur une chaîne de caractères précisant la classe sur laquelle elle va travailler.

Exemple (code Java 1.1) :

```
import java.util.*;
import java.lang.reflect.*;
public class ClasseInspecteur {
  private Class classe;
  private String nomClasse;
  public ClasseInspecteur(String nomClasse) {
    this.nomClasse = nomClasse;
    try {
      classe = Class.forName(nomClasse);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Ce chapitre contient plusieurs sections :

- La classe Class
- La recherche des informations sur une classe
- La définition dynamique d'objets

22.1. La classe Class

Les instances de la classe Class sont des objets représentant les classes du langage. Il y aura une instance représentant chaque classe utilisée : par exemple la classe String, la classe Frame, la classe Class, etc ... . Ces instances sont crées automatiquement par la machine virtuelle lors du chargement de la classe. Il est ainsi possible de connaître les caractéristiques d'une classe de façon dynamique en utilisant les méthodes de la classe Class. Les applications telles que les debogueurs, les inspecteurs d'objets et les environnements de développement doivent faire une analyse des objets qu'ils manipulent en utilisant ces mécanismes.

La classe Class est définie dans le package java.lang.

La classe Class permet :

* de décrire une classe ou une interface par introspection : obtenir son nom, sa classe mère, la liste de ces méthodes, de ses variables de classe, de ses constructeurs et variables d'instances, etc ...
* d'agir sur une classe en envoyant, à un objet Class des messages comme à tout autre objet. Par exemple, créer dynamiquement à partir d'un objet Class une nouvelle instance de la classe représentée

22.1.1. L'obtention d'un objet à partir de la classe Class

La classe Class ne possède pas de constructeur public mais il existe plusieurs façons d'obtenir un objet de la classe Class.

22.1.1.1. La détermination de la classe d'un objet

La méthode getClass() définit dans la classe Object renvoie une instance de la classe Class. Par héritage, tout objet java dispose de cette méthode.

Exemple (code Java 1.1) :

```
package introspection;

public class TestGetClass {
   public static void main(java.lang.String[] args) {
      String chaine = "test";
      Class classe = chaine.getClass();
      System.out.println("classe de l'objet chaine = "+classe.getName());
   }
}
```

Résultat :

```
classe de l'objet chaine = java.lang.String
```

22.1.1.2. L'obtention d'un objet Class à partir d'un nom de classe

La classe Class possède une méthode statique forName() qui permet à partir d'une chaîne de caractères désignant une classe d'instancier un objet de cette classe et de renvoyer un objet de la classe Class pour cette classe.

Cette méthode peut lever l'exception ClassNotFoundException.

Exemple (code Java 1.1) :

```
public class TestForName {
   public static void main(java.lang.String[] args) {
      try {
         Class classe = Class.forName("java.lang.String");
         System.out.println("classe de l'objet chaine = "+classe.getName());
      }
      catch(Exception e) {
         e.printStackTrace();
      }
   }
}
```

Résultat :

```
classe de l'objet chaîne = java.lang.String
```

22.1.1.3. Une troisième façon d'obtenir un objet Class

Il est possible d'avoir un objet de la classe Class en écrivant type.class ou type est le nom d'une classe.

Exemple (code Java 1.1) :

```
package introspection;
public class TestClass {
   public static void main(java.lang.String[] args) {
      Class c = Object.class;
      System.out.println("classe de Object  = "+c.getName());
   }
}
```

Résultat :

```
classe de Object  = java.lang.Object
```

22.1.2. Les méthodes de la classe Class

La classe Class fournie de nombreuses méthodes pour obtenir des informations sur la classe qu'elle représente. Voici les principales méthodes :

Méthodes	Rôle
static Class forName(String)	Instancie un objet de la classe dont le nom est fourni en paramètre et renvoie un objet Class la représentant
Class[] getClasses()	Renvoie les classes et interfaces publiques qui sont membres de la classe
Constructor[] getConstructors()	Renvoie les constructeurs publics de la classe
Class[] getDeclaredClasses()	Renvoie un tableau des classes définies comme membre dans la classe
Constructor[] getDeclaredConstructors()	Renvoie tous les constructeurs de la classe
Field[] getDeclaredFields()	Renvoie un tableau de tous les attributs définis dans la classe
Method getDeclaredMethods()	Renvoie un tableau de toutes les méthodes
Field getFields()	Renvoie un tableau des attributs publics
Class[] getInterfaces()	Renvoie un tableau des interfaces implémentées par la classe
Method getMethod()	Renvoie un tableau des méthodes publiques de la classe incluant celles héritées
int getModifiers()	Renvoie un entier qu'il faut décoder pour connaître les modificateurs de la classe
Package getPackage()	Renvoie le package de la classe
Classe getSuperClass()	Renvoie la classe mère de la classe
boolean isArray()	Indique si la classe est un tableau
boolean IsInterface()	Indique si la classe est une interface
Object newInstance()	Permet de créer une nouvelle instance de la classe

22.2. La recherche des informations sur une classe

En utilisant les méthodes de la classe Class, il est possible d'obtenir quasiment toutes les informations sur une classe.

22.2.1. La recherche de la classe mère d'une classe

La classe Class possède une méthode getSuperClass() qui retourne un objet de la classe Class représentant la classe mère si elle existe sinon elle retourne null.

Pour obtenir toute la hiérarchie d'une classe il suffit d'appeler successivement cette méthode sur l'objet qu'elle a retourné.

Exemple (code Java 1.1) : méthode qui retourne un vecteur contenant les classes mères

```
public Vector getClassesParentes() {

  Vector cp = new Vector();

  Class sousClasse = classe;
  Class superClasse;

  cp.add(sousClasse.getName());
  superClasse = sousClasse.getSuperclass();
  while (superClasse != null) {
    cp.add(0,superClasse.getName());
    sousClasse = superClasse;
    superClasse = sousClasse.getSuperclass();
  }
  return cp;
}
```

22.2.2. La recherche des modificateurs d'une classe

La classe Class possède une méthode getModifiers() qui retourne un entier représentant les modificateurs de la classe. Pour décoder cette valeur, la classe Modifier possède plusieurs méthodes qui attendent cet entier en paramètre et qui retourne un booléen selon leur fonction : isPublic(), isAbstract(), isFinal(), ...

La classe Modifier ne contient que des constantes et des méthodes statiques qui permettent de déterminer les modificateurs d'accès :

Méthode	Rôle
boolean isAbstract(int)	Renvoie true si le paramètre contient le modificateur abstract
boolean isFinal(int)	Renvoie true si le paramètre contient le modificateur final
boolean isInterface(int)	Renvoie true si le paramètre contient le modificateur interface
boolean isNative(int)	Renvoie true si le paramètre contient le modificateur native
boolean isPrivate(int)	Renvoie true si le paramètre contient le modificateur private
boolean isProtected(int)	Renvoie true si le paramètre contient le modificateur protected
boolean isPublic(int)	Renvoie true si le paramètre contient le modificateur public
boolean isStatic(int)	Renvoie true si le paramètre contient le modificateur static
boolean isSynchronized(int)	Renvoie true si le paramètre contient le modificateur synchronized
boolean isTransient(int)	Renvoie true si le paramètre contient le modificateur transient
boolean isVolatile(int)	Renvoie true si le paramètre contient le modificateur volatile

Ces méthodes étant static il est inutile d'instancier un objet de type Modifier pour utiliser ces méthodes.

Exemple (code Java 1.1) :

```
public Vector getModificateurs() {

  Vector cp = new Vector();
  int m = classe.getModifiers();
  if (Modifier.isPublic(m))
    cp.add("public");
  if (Modifier.isAbstract(m))
    cp.add("abstract");
  if (Modifier.isFinal(m))
    cp.add("final");

  return cp;
}
```

22.2.3. La recherche des interfaces implémentées par une classe

La classe Class possède une méthode getInterfaces() qui retourne un tableau d'objet de type Class contenant les interfaces implémentées par la classe.

Exemple (code Java 1.1) :

```
public Vector getInterfaces() {

  Vector cp = new Vector();

  Class[] interfaces = classe.getInterfaces();
  for (int i = 0; i < interfaces.length; i++) {
    cp.add(interfaces[i].getName());
  }

  return cp;
}
```

22.2.4. La recherche des champs publics

La classe Class possède une méthode getFields() qui retourne les attributs public de la classe. Cette méthode retourne un tableau d'objet de type Field.

La classe Class possède aussi une méthode getField() qui attend en paramètre un nom d'attribut et retourne un objet de type Field si celui ci est défini dans la classe ou dans une de ses classes mères. Si la classe ne contient pas d'attribut dont le nom correspond au paramètre fourni, la méthode getField() lève une exception de la classe NoSuchFieldException.

La classe Field représente un attribut d'une classe ou d'une interface et permet d'obtenir des informations cet attribut. Elle possède plusieurs méthodes :

Méthode	Rôle
String getName()	Retourne le nom de l'attribut
Class getType()	Retourne un objet de type Class qui représente le type de l'attribut
Class getDeclaringClass()	Retourne un objet de type Class qui représente la classe qui définie l'attribut
int getModifiers()	Retourne un entier qui décrit les modificateurs d'accès. Pour les connaître précisément il faut utiliser les méthodes static de la classe Modifier.

Object get(Object)	Retourne la valeur de l'attribut pour l'instance de l'objet fourni en paramètre. Il existe aussi plusieurs méthodes getXXX() ou XXX représente un type primitif et qui la renvoie la valeur dans ce type.

Exemple (code Java 1.1) :

```
public Vector getChampsPublics() {

  Vector cp = new Vector();

  Field[] champs = classe.getFields();
  for (int i = 0; i < champs.length; i++)
    cp.add(champs[i].getType().getName()+" "+champs[i].getName());
  return cp;
}
```

22.2.5. La recherche des paramètres d'une méthode ou d'un constructeur

L'exemple ci dessous présente une méthode qui permet de formater sous forme de chaîne de caractères les paramètres d'une méthode fournis sous la forme d'un tableau d'objets de type Class.

Exemple (code Java 1.1) :

```
private String rechercheParametres(Class[] classes) {

  StringBuffer param = new StringBuffer("(");

  for (int i = 0; i < classes.length; i ++) {

    param.append(formatParametre(classes[i].getName()));
    if (i < classes.length - 1)
      param.append(", ");

  }
  param.append(")");

  return param.toString();

}
```

La méthode getName() de la classe Class renvoie une chaîne de caractères formatée qui précise le type de la classe. Ce type est représenté par une chaîne de caractères qu'il faut décoder pour obtenir le type.

Si le type de la classe est un tableau alors la chaîne commence par un nombre de caractère '[' correspondant à la dimension du tableau.

Ensuite la chaîne contient un caractère qui précise un type primitif ou un objet. Dans le cas d'un objet, le nom de la classe de l'objet avec son package complet est contenu dans la chaîne suivi d'un caractère ';'.

Caractère	Type
B	byte
C	char
D	double
F	float
I	int
J	long

Lclassname;	classe ou interface
S	short
Z	boolean

Exemple :

La méthode getName() de la classe Class représentant un objet de type float[10][5] renvoie « [[F »

Pour simplifier les traitements, la méthode formatParametre() ci dessous retourne une chaîne de caractères qui décode le contenu de la chaîne retournée par la méthode getName() de la classe Class.

Exemple :

```
private String formatParametre(String s) {

  if (s.charAt(0) == '[') {

    StringBuffer param = new StringBuffer("");

    int dimension = 0;
    while (s.charAt(dimension) == '[') dimension++;

    switch(s.charAt(dimension)) {
      case 'B' : param.append("byte");break;
      case 'C' : param.append("char");break;
      case 'D' : param.append("double");break;
      case 'F' : param.append("float");break;
      case 'I' : param.append("int");break;
      case 'J' : param.append("long");break;
      case 'S' : param.append("short");break;
      case 'Z' : param.append("boolean");break;
      case 'L' : param.append(s.substring(dimension+1,s.indexOf(";")));
    }

    for (int i =0; i < dimension; i++)
      param.append("[]");

    return param.toString();
  }
  else return s;

}
```

22.2.6. La recherche des constructeurs de la classe

La classe Class possède une méthode getConstructors() qui retourne un tableau d'objet de type Constructor contenant les constructeurs de la classe.

La classe Constructor représente un constructeur d'une classe. Elle possède plusieurs méthodes :

Méthode	Rôle
String getName()	Retourne le nom du constructeur
Class[] getExceptionTypes()	Retourne un tableau de type Class qui représente les exceptions qui peuvent être propagées par le constructeur
Class[] getParametersType()	Retourne un tableau de type Class qui représente les paramètres du constructeur
int getModifiers()	Retourne un entier qui décrit les modificateurs d'accès. Pour les connaître précisément il

	faut utiliser les méthodes static de la classe Modifier.
Object newInstance(Object[])	Instancie un objet en utilisant le constructeur avec les paramètres fournis à la méthode

Exemple (code Java 1.1) :

```
public Vector getConstructeurs() {

  Vector cp = new Vector();
  Constructor[] constructeurs = classe.getConstructors();
  for (int i = 0; i < constructeurs.length; i++) {
    cp.add(rechercheParametres(constructeurs[i].getParameterTypes()));

  }

  return cp;
}
```

L'exemple ci dessus utilise la méthode rechercherParamètres() définie précédemment pour simplifier les traitements.

22.2.7. La recherche des méthodes publiques

Pour consulter les méthodes d'un objet, il faut obtenir sa classe et lui envoyer le message getMethod(), qui renvoie les méthodes publiques qui sont déclarées dans la classe et qui sont héritées des classes mères.

Elle renvoie un tableau d'instances de la classe Method du package java.lang.reflect.

Une méthode est caractérisée par un nom, une valeur de retour, une liste de paramètres, une liste d'exceptions et une classe d'appartenance.

La classe Method contient plusieurs méthodes :

Méthode	Rôle
Class[] getParameterTypes	Renvoie un tableau de classes représentant les paramètres.
Class getReturnType	Renvoie le type de la valeur de retour de la méthode.
String getName()	Renvoie le nom de la méthode
int getModifiers()	Renvoie un entier qui représente les modificateurs d'accès
Class[] getExceptionTypes	Renvoie un tableau de classes contenant les exceptions propagées par la méthode
Class getDeclaringClass[]	Renvoie la classe qui définit la méthode

Exemple (code Java 1.1) :

```
public Vector getMethodesPubliques() {

  Vector cp = new Vector();
  Method[] methodes = classe.getMethods();
  for (int i = 0; i < methodes.length; i++) {
    StringBuffer methode = new StringBuffer();

    methode.append(formatParametre(methodes[i].getReturnType().getName()));
    methode.append(" ");
    methode.append(methodes[i].getName());
    methode.append(rechercheParametres(methodes[i].getParameterTypes()));

    cp.add(methode.toString());

  }
  return cp;
}
```

L'exemple ci dessus utilise les méthodes formatParametre() et rechercherParametres() définies précédemment pour simplifier les traitements.

22.2.8. La recherche de toutes les méthodes

Pour consulter toutes les méthodes d'un objet, il faut obtenir sa classe et lui envoyer le message getDeclareMethods(), qui renvoie toutes les méthodes qui sont déclarées dans la classe et qui sont héritées des classes mères quelque soit leur accessibilité.

Elle renvoie un tableau d'instances de la classe Method du package java.lang.reflect.

Exemple :

```
public Vector getMethodes() {

  Vector cp = new Vector();
  Method[] methodes = classe.getDeclareMethods();
  for (int i = 0; i < methodes.length; i++) {
    StringBuffer methode = new StringBuffer();

    methode.append(formatParametre(methodes[i].getReturnType().getName()));
    methode.append(" ");
    methode.append(methodes[i].getName());
    methode.append(rechercheParametres(methodes[i].getParameterTypes()));

    cp.add(methode.toString());

  }

  return cp;
}
```

L'exemple ci dessus utilise les méthodes formatParametre() et rechercherParametres() définies précédemment pour simplifier les traitements.

22.3. La définition dynamique d'objets

22.3.1. La définition d'objets grâce à la classe Class

Cette section sera développée dans une version future de ce document

22.3.2. L'exécution dynamique d'une méthode

Cette section sera développée dans une version future de ce document

23. L'appel de méthodes distantes : RMI

RMI (Remote Method Invocation) est une technologie développée et fournie par Sun à partir du JDK 1.1 pour permettre de mettre en oeuvre facilement des objets distribués.

Ce chapitre contient plusieurs sections :

- La présentation et l'architecture de RMI
- Les différentes étapes pour créer un objet distant et l'appeler avec RMI
- Le développement coté serveur
- Le développement coté client
- La génération des classes stub et skeleton
- La mise en oeuvre des objets RMI

23.1. La présentation et l'architecture de RMI

Le but de RMI est de permettre l'appel, l'exécution et le renvoi du résultat d'une méthode exécutée dans une machine virtuelle différente de celle de l'objet l'appelant. Cette machine virtuelle peut être sur une machine différente pourvu qu'elle soit accessible par le réseau.

La machine sur laquelle s'exécute la méthode distante est appelée serveur.

L'appel coté client d'une telle méthode est un peu plus compliqué que l'appel d'une méthode d'un objet local mais il reste simple. Il consiste à obtenir une référence sur l'objet distant puis à simplement appeler la méthode à partir de cette référence.

La technologie RMI se charge de rendre transparente la localisation de l'objet distant, son appel et le renvoi du résultat.

En fait, elle utilise deux classes particulières, le stub et le skeleton, qui doivent être générées avec l'outil rmic fourni avec le JDK.

Le stub est une classe qui se situe côté client et le skeleton est son homologue coté serveur. Ces deux classes se chargent d'assurer tous les mécanismes d'appel, de communication, d'exécution, de renvoi et de réception du résultat.

23.2. Les différentes étapes pour créer un objet distant et l'appeler avec RMI

Le développement coté serveur se compose de :

- La définition d'une interface qui contient les méthodes qui peuvent être appelées à distance
- L'écriture d'une classe qui implémente cette interface
- L'écriture d'une classe qui instanciera l'objet et l'enregistrera en lui affectant un nom dans le registre de nom RMI (RMI Registry)

Le développement côté client se compose de :

* L'obtention d'une référence sur l'objet distant à partir de son nom
* L'appel à la méthode à partir de cette référence

Enfin, il faut générer les classes stub et skeleton en exécutant le programme rmic avec le fichier source de l'objet distant

23.3. Le développement coté serveur

23.3.1. La définition d'une interface qui contient les méthodes de l'objet distant

L'interface à définir doit hériter de l'interface java.rmi.Remote. Cette interface ne contient aucune méthode mais indique simplement que l'interface peut être appelée à distance.

L'interface doit contenir toutes les méthodes qui seront susceptibles d'être appelées à distance.

La communication entre le client et le serveur lors de l'invocation de la méthode distante peut échouer pour diverses raisons telles qu'un crash du serveur, une rupture de la liaison, etc ...

Ainsi chaque méthode appelée à distance doit déclarer qu'elle est en mesure de lever l'exception java.rmi.RemoteException.

Exemple (code Java 1.1) :

```
package test_rmi;

import java.rmi.*;

public interface Information extends Remote {

   public String getInformation() throws RemoteException;

}
```

23.3.2. L'écriture d'une classe qui implémente cette interface

Cette classe correspond à l'objet distant. Elle doit donc implémenter l'interface définie et contenir le code nécessaire.

Cette classe doit obligatoirement hériter de la classe UnicastRemoteObject qui contient les différents traitements élémentaires pour un objet distant dont l'appel par le stub du client est unique. Le stub ne peut obtenir qu'une seule référence sur un objet distant héritant de la classe UnicastRemoteObject. On peut supposer qu'une future version de RMI sera capable de faire du MultiCast, permettant à RMI de choisir parmi plusieurs objets distants identiques la référence à fournir au client.

La hiérarchie de la classe UnicastRemoteObject est :

java.lang.Object

  java.rmi.Server.RemoteObject

    java.rmi.Server.RemoteServer

      java.rmi.Server.UnicastRemoteObject

Comme indiqué dans l'interface, toutes les méthodes distantes doivent indiquer qu'elles peuvent lever l'exception RemoteException mais aussi le constructeur de la classe. Ainsi, même si le constructeur ne contient pas de code il doit être redéfini pour inhiber la génération du constructeur par défaut qui ne lève pas cette exception.

Exemple (code Java 1.1) :

```
package test_rmi;

import java.rmi.*;

import java.rmi.server.*;

public class TestRMIServer extends UnicastRemoteObject implements Information {

   protected TestRMIServer() throws RemoteException {
      super();
   }

   public String getInformation()throws RemoteException {
      return "bonjour";
   }

}
```

23.3.3. L'écriture d'une classe pour instancier l'objet et l'enregistrer dans le registre

Ces opérations peuvent être effectuées dans la méthode main d'une classe dédiée ou dans la méthode main de la classe de l'objet distant. L'intérêt d'une classe dédiée et qu'elle permet de regrouper toutes ces opérations pour un ensemble d'objets distants.

La marche à suivre contient trois étapes :

- la mise en place d'un security manager dédié qui est facultative
- l'instanciation d'un objet de la classe distante
- l'enregistrement de la classe dans le registre de nom RMI en lui donnant un nom

23.3.3.1. La mise en place d'un security manager

Cette opération n'est pas obligatoire mais elle est recommandée en particulier si le serveur doit charger des classes qui ne sont pas sur le serveur. Sans security manager, il faut obligatoirement mettre à la disposition du serveur toutes les classes dont il aura besoin (Elles doivent être dans le CLASSPATH du serveur). Avec un security manager, le serveur peut charger dynamiquement certaines classes.

Cependant, le chargement dynamique de ces classes peut poser des problèmes de sécurité car le serveur va exécuter du code d'une autre machine. Cet aspect peut conduire à ne pas utiliser de security manager.

Exemple (code Java 1.1) :

```
public static void main(String[] args) {
   try {
      System.out.println("Mise en place du Security Manager ...");
      System.setSecurityManager(new java.rmi.RMISecurityManager());
   } catch (Exception e) {
      System.out.println("Exception capturée: " + e.getMessage());
   }
}
```

23.3.3.2. L'instanciation d'un objet de la classe distante

Cette opération est très simple puisqu'elle consiste simplement en la création d'un objet de la classe de l'objet distant

Exemple (code Java 1.1) :

```
public static void main(String[] args) {

   try {
      System.out.println("Mise en place du Security Manager ...");
      System.setSecurityManager(new java.rmi.RMISecurityManager());

      TestRMIServer testRMIServer = new TestRMIServer();

   } catch (Exception e) {

      System.out.println("Exception capturée: " + e.getMessage());

   }

}
```

23.3.3.3. L'enregistrement dans le registre de nom RMI en lui donnant un nom

La dernière opération consiste à enregistrer l'objet créé dans le registre de nom en lui affectant un nom. Ce nom est fourni au registre sous forme d'une URL constitué du préfix rmi://, du nom du seveur (hostname) et du nom associé à l'objet précédé d'un slash.

Le nom du serveur peut être fourni « en dur » sous forme d'une constante chaîne de caractères ou peut être dynamiquement obtenu en utilisant la classe InetAddress pour une utilisation en locale.

C'est ce nom qui sera utilisé dans une URL par le client pour obtenir une référence sur l'objet distant.

L'enregistrement se fait en utilisant la méthode rebind de la classe Naming. Elle attend en paramètre l'URL du nom de l'objet et l'objet lui même.

Exemple (code Java 1.1) :

```
public static void main(String[] args) {

   try {

      System.out.println("Mise en place du Security Manager ...");
      System.setSecurityManager(new java.rmi.RMISecurityManager());

      TestRMIServer testRMIServer = new TestRMIServer();

      System.out.println("Enregistrement du serveur");

      Naming.rebind("rmi://"+java.net.InetAddress.getLocalHost()+
         "/TestRMI",testRMIServer);

      // Naming.rebind(";rmi://localhost/TestRMI", testRMIServer);

      System.out.println("Serveur lancé");

   } catch (Exception e) {
      System.out.println("Exception capturée: " + e.getMessage());
   }

}
```

23.3.3.4. Le lancement dynamique du registre de nom RMI

Sur le serveur, le registre de nom RMI doit s'exécuter avant de pouvoir enregistrer un objet ou obtenir une référence.

Ce registre peut être lancé en tant qu'application fournie dans le JDK (rmiregistry) comme indiqué dans un chapitre suivant ou être lancé dynamiquement dans la classe qui enregistre l'objet. Ce lancement ne doit avoir lieu qu'une seule et unique fois. Il peut être intéressant d'utiliser ce code si l'on crée une classe dédiée à l'enregistrement des objets distants.

Le code pour exécuter le registre est la méthode createRegistry de la classe java.rmi.registry.LocateRegistry. Cette méthode attend en paramètre un numéro de port.

Exemple (code Java 1.1) :

```
public static void main(String[] args) {

   try {

      java.rmi.registry.LocateRegistry.createRegistry(1099);

      System.out.println("Mise en place du Security Manager ...");
      System.setSecurityManager(new java.rmi.RMISecurityManager());

      ...

   }
}
```

23.4. Le développement coté client

L'appel d'une méthode distante peut se faire dans une application ou dans une applet.

23.4.1. La mise en place d'un security manager

Comme pour le coté serveur, cette opération est facultative.

Le choix de la mise en place d'un sécurity manager côté client suit des règles identiques à celui du côté serveur. Sans son utilisation, il est nécessaire de mettre dans le CLASSPATH du client toutes les classes nécessaires dont la classe stub.

Exemple (code Java 1.1) :

```
public static void main(String[] args) {

   System.setSecurityManager(new RMISecurityManager());

}
```

23.4.2. L'obtention d'une référence sur l'objet distant à partir de son nom

Pour obtenir une référence sur l'objet distant à partir de son nom, il faut utiliser la méthode statique lookup() de la classe Naming.

Cette méthode attend en paramètre une URL indiquant le nom qui référence l'objet distant. Cette URL est composée de plusieurs éléments : le préfix rmi://, le nom du serveur (hostname) et le nom de l'objet tel qu'il a été enregistré dans le registre précédé d'un slash.

Il est préférable de prévoir le nom du serveur sous forme de paramètres de l'application ou de l'applet pour plus de souplesse.

La méthode lookup() va rechercher dans le registre du serveur l'objet et retourner un objet stub. L'objet retourné est de la classe Remote (cette classe est la classe mère de tous les objets distants).

Si le nom fourni dans l'URL n'est pas référencé dans le registre, la méthode lève l'exception NotBoundException.

Exemple (code Java 1.1) :

```
public static void main(String[] args) {

   System.setSecurityManager(new RMISecurityManager());

   try {

      Remote r = Naming.lookup("rmi://vaio/127.0.0.1/TestRMI");

   } catch (Exception e) {
   }

}
```

23.4.3. L'appel à la méthode à partir de la référence sur l'objet distant

L'objet retourné étant de type Remote, il faut réaliser un cast vers l'interface qui définit les méthodes de l'objet distant. Pour plus de sécurité, on vérifie que l'objet retourné est bien une instance de cette interface.

Un fois le cast réalisé, il suffit simplement d'appeler la méthode.

Exemple (code Java 1.1) :

```
public static void main(String[] args) {

   System.setSecurityManager(new RMISecurityManager());

   try {

      Remote r = Naming.lookup("rmi://vaio/127.0.0.1/TestRMI");

      if (r instanceof Information) {
         String s = ((Information) r).getInformation();
         System.out.println("chaine renvoyée = " + s);
      }

   } catch (Exception e) {
   }
}
```

23.4.4. L'appel d'une méthode distante dans une applet

L'appel d'une méthode distante est la même dans une application et dans une applet.

Seule la mise en place d'un security manager dédié dans les applets est inutile car elles utilisent déjà un sécurity manager (AppletSecurityManager) qui autorise le chargement de classes distantes.

Exemple (code Java 1.1) :

```
package test_rmi;

import java.applet.*;
import java.awt.*;
import java.rmi.*;

public class AppletTestRMI extends Applet {

   private String s;
```

```
   public void init() {

      try {
         Remote r = Naming.lookup("rmi://vaio/127.0.0.1/TestRMI");

         if (r instanceof Information) {
            s = ((Information) r).getInformation();
         }
      } catch (Exception e) {
      }

   }

   public void paint(Graphics g) {
      super.paint(g);
      g.drawString("chaine retournée = "+s,20,20);
   }

}
```

23.5. La génération des classes stub et skeleton

Pour générer ces classes, il suffit d'utiliser l'outil rmic fourni avec le JDK en lui donnant en paramètre le nom de la classe.

Attention la classe doit avoir été compilée : rmic à besoin du fichier .class.

Exemple (code Java 1.1) :

```
rmic test_rmi.TestRMIServer
```

rmic va générer et compiler les classes stub et skeleton respectivement sous le nom TestRMIServer_Stub.class et TestRMIServer_Skel.class

23.6. La mise en oeuvre des objets RMI

La mise en oeuvre et l'utilisation d'objet distant avec RMI nécessite plusieurs étapes :

1. Démarrer le registre RMI sur le serveur soit en utilisant le programme rmiregistry livré avec le JDK soit en exécutant une classe qui effectue le lancement.
2. exécuter la classe qui instancie l'objet distant et l'enregistre dans le serveur de nom RMI
3. Lancer l'application ou l'applet pour tester.

23.6.1. Le lancement du registre RMI

La commande rmiregistry est fournie avec le JDK.

Il faut la lancer en tache de fond :

Sous Unix : rmiregistry&

Sous Windows : start rmiregistry

Ce registre permet de faire correspondre un objet à un nom et inversement. C'est lui qui est sollicité lors d'un appel aux méthodes Naming.bind() et Naming.lookup()

23.6.2. L'instanciation et l'enregistrement de l'objet distant

Il faut exécuter la classe qui va instancier l'objet distant et l'enregistrer sous son nom dans le registre précédemment lancé.

Pour ne pas avoir de problème, il faut s'assurer que toutes les classes utiles (la classe de l'objet distant, l'interface qui définit les méthodes, le skeleton) sont présentes dans un répertoire défini dans la variable CLASSPATH.

23.6.3. Le lancement de l'application cliente

La suite de cette section sera développée dans une version future de ce document

24. L'internationalisation

La localisation consiste à adapter un logiciel pour s'adapter aux caractéristiques locales de l'environnement d'exécution telles que la langue. Le plus gros du travail consiste à traduire toutes les phrases et les mots. Les classes nécessaires sont incluses dans le packetage java.util.

Ce chapitre contient plusieurs sections :

- Les objets de type Locale
- La classe ResourceBundle
- Des chemins guidés pour réaliser la localisation : cette section propose plusieurs solutions pour réalisation l'internationalisation.

24.1. Les objets de type Locale

Un objet de type Locale identifie une langue et un pays donné.

24.1.1. La création d'un objet Locale

Exemple (code Java 1.1) :

```
locale_US = new
Locale("en","US") ;
locale_FR = new Locale("fr","FR");
```

Le premier paramètre est le code langue (deux caractères minuscules conformes à la norme ISO-639 : exemple "de" pour l'allemand, "en" pour l'anglais, "fr" pour le français, etc ...)

Le deuxième paramètre est le code pays (deux caractères majuscules conformes à la norme ISO-3166 : exemple : "DE" pour l'Allemagne, "FR" pour la France,"US" pour les Etats Unis, etc ...). Ce paramètre est obligatoire : si le pays n'a pas besoin d'être précisé, il faut fournir une chaîne vide.

Exemple (code Java 1.1) :

```
Locale locale = new Locale("fr", "");
```

Un troisième paramètre peut permettre de préciser d'avantage la localisation par exemple la plateforme d'exécution (il ne respecte aucun standard car il ne sera défini que dans l'application qui l'utilise) :

Exemple (code Java 1.1) :

```
Locale locale_unix = new Locale("fr","FR", "UNIX");
Locale locale_windows = new Locale("fr","FR", "WINDOWS");
```

Ce troisième paramètre est optionnel.

La classe Locale définit des constantes pour certaines langues et pays :

Exemple (code Java 1.1) : ces deux lignes sont équivalentes

```
Locale locale_1 = Locale.JAPAN;
Locale locale_2 = new Locale("ja", "JP");
```

Lorsque l'on précise une constante représentant une langue alors le code pays n'est pas défini.

Exemple (code Java 1.1) : ces deux lignes sont équivalentes

```
Locale locale_3 = Locale.JAPANESE;
Locale locale_2 = new Locale("ja", "");
```

Il est possible de rendre la création d'un objet Locale dynamique :

Exemple (code Java 1.1) :

```
static public void main(String[] args) {
   String langue = new String(args[0]);
   String pays = new String(args[1]);
   locale = new Locale(langue, pays);
}
```

Cet objet ne sert que d'identifiant qu'il faut passer à des objets de type ResourceBundle par exemple qui eux possèdent le nécessaire pour réaliser la localisation. En fait, la création d'un objet Locale pour un pays donné ne signifie pas que l'on va pouvoir l'utiliser.

24.1.2. L'obtention de la liste des Locales disponibles

La méthode getAvailableLocales() permet de connaître la liste des Locales reconnues par une classe sensible à l'internationalisation

Exemple (code Java 1.1) : avec la classe DateFormat

```
import java.util.*;
import java.text.*;

public class Available {
   static public void main(String[] args) {
      Locale liste[] =
      DateFormat.getAvailableLocales();
      for (int i = 0; i < liste.length; i++)
      {
         System.out.println(liste[i].toString());
         // toString retourne le code langue et le code pays séparé d'un souligné
      }
   }
}
```

La méthode Locale.getDisplayName() peut être utilisée à la place de toString pour obtenir le nom du code langue et du code pays.

24.1.3. L'utilisation d'un objet Locale

Il n'est pas obligatoire de se servir du même objet Locale avec les classes sensibles à l'internationalisation.

Cependant la plupart des applications utilisent l'objet Locale par défaut initialisé par la machine virtuelle avec les paramètres de la machine hôte. La méthode Locale.getDefault() permet de connaître l'objet Locale par défaut.

24.2. La classe ResourceBundle

Il est préférable de définir un ResourceBundle pour chaque catégorie d'objet (exemple un par fenêtre) : ceci rend le code plus clair et plus facile à maintenir, évite d'avoir des ResourceBundle trop importants et réduit l'espace mémoire utilisé car chaque ressource n'est chargée que lorsque l'on en a besoin.

24.2.1. La création d'un objet ResourceBundle

Conceptuellement, chaque ResourceBundle est un ensemble de sous classes qui partage la même racine de nom.

Exemple :

```
TitreBouton
TitreBouton_de
TitreBouton_en_GB
TitreBouton_fr_FR_UNIX
```

Pour sélectionner le ResourceBundle approprié il faut utiliser la méthode ResourceBundle.getBundle().

Exemple (code Java 1.1) :

```
Locale locale = new Locale("fr", "FR");
ResourceBundle messages = ResourceBundle.getBundle("TitreBouton", locale);
```

Le premier argument contient le type d'objet à utiliser (la racine du nom de cet objet).

Le second argument de type Locale permet de déterminer quel fichier sera utilisé : il ajoute le code pays et le code langue séparé par un souligné à la racine du nom.

Si la classe désignée par l'objet Locale n'existe par, alors getBundle recherche celle qui se rapproche le plus. L'ordre de recherche sera le suivant :

Exemple :

```
TitreBouton_fr_CA_UNIX
TitreBouton_fr_FR
TitreBouton_fr
TitreBouton_en_US
TitreBouton_en
TitreBouton
```

Si aucune n'est trouvée alors getBundle lève une exception de type MissingResourceException.

24.2.2. Les sous classes de ResourceBundle

La classe abstraite ResourceBundle possède deux sous classes : PropertyResourceBundle et ListResourceBundle.

La classe ResourceBundle est une classe flexible : le changement de l'utilisation d'un PropertyResourceBundle en ListResourceBundle se fait sans impact sur le code. La méthode getBundle() recherche le ResourceBundle désiré qu'il soit dans un fichier .class ou propriétés

24.2.2.1. L'utilisation de PropertyResourceBundle

Un PropertyResourceBundle est rattaché à un fichier propriétés. Ces fichiers propriétés ne font pas partie du code source java. Ils ne peuvent contenir que des chaines de caractères. Pour stocker d'autres objets, il faut utiliser des objets ListResourceBundle.

La création d'un fichier propriétés est simple : c'est un fichier texte qui contient des paires clé-valeur. La clé et la valeur sont séparées par un signe =. Chaque paire doit être sur une ligne séparée.

Exemple (code Java 1.1) :

```
texte_suivant = suivant
texte_precedent = precedent
```

Le nom du fichier propriétés par défaut se compose de la racine du nom suivi de l'extension .properties.

Exemple : TitreBouton.properties.

Dans une autre langue, anglais par exemple, le fichier s'appellerait : TitreBouton_en.properties

Exemple (code Java 1.1) :

```
texte_suivant = next
texte_precedent = previous
```

Les clés sont les mêmes, seule la traduction change.

Le nom de fichier TitreBouton_fr_FR.properties contient la racine (Titrebouton), le code langue (fr) et le code pays (FR).

24.2.2.2. L'utilisation de ListResourceBundle

La classe ListResourceBundle gère les ressources sous forme de liste encapsulée dans un objet. Chaque ListResourceBundle est donc rattaché à un fichier .class. On peut y stocker n'importe quel objet spécifique à la localisation.

Les objets ListResourceBundle contiennent des paires clé-valeur. La clé doit être une chaîne qui caractérise l'objet. La valeur est un objet de n'importe quelle classe.

Exemple (code Java 1.1) :

```
class TitreBouton_fr extends ListResourceBundle {
   public Object[][] getContents() {
      return contents;
   }
   static final Object[][] contents = {
      {"texte_suivant", "Suivant"},
      {"texte_precedent", "Precedent"},
   };
}
```

24.2.3. L'obtention d'un texte d'un objet ResourceBundle

La méthode getString() retourne la valeur de la clé précisée en paramètre.

Exemple (code Java 1.1) :

```
String message_1 = messages.getString("texte_suivant");
String message_2 = TitreBouton.getString("texte_suivant");
```

24.3. Des chemins guidés pour réaliser la localisation

24.3.1. L'utilisation d'un ResourceBundle avec un fichier propriétés

Il faut toujours créer le fichier propriété par défaut. Le nom de ce fichier commence avec le nom de base du ResourceBundle et se termine avec le suffix .properties

Exemple (code Java 1.1) :

```
#Exemple de fichier propriété par défaut (TitreBouton.properties)
texte1 = suivant
texte2 = precedent
texte3 = quitter
```

Les lignes de commentaires commencent par un #. Les autres lignes contiennent les paires clé-valeur. Une fois le fichier défini, il ne faut plus modifier la valeur de la clé qui pourrait être appelée dans un programme.

Pour ajouter le support d'autre langue, il faut créer des fichiers propriétés supplémentaires qui contiendront les traductions.
Le fichier est le même, seul le texte contenu dans la valeur change (la valeur de la clé doit être identique).

Exemple (code Java 1.1) :

```
#Exemple de fichier propriété en anglais (TitreBouton_en.properties)
texte1 = next
texte2 = previous
texte3 = quit
```

Lors de la programmation, il faut créer un objet Locale. Il est possible de créer un tableau qui contient la liste des Locale disponibles en fonction des fichiers propriétés créés.

Exemple (code Java 1.1) :

```
Locale[] locales = { Locale.GERMAN, Locale.ENGLISH };
```

Dans cet exemple, l'objet Locale.ENGLISH correspond au fichier TitreBouton_en.properties. L'objet Locale.GERMAN ne possédant pas de fichier propriétés défini, le fichier par défaut sera utilisé.

Il faut créer l'objet ResourceBundle en invoquant la méthode getBundle de l'objet Locale.

Exemple (code Java 1.1) :

```
ResourceBundle titres = ResourceBundle.getBundle("TitreBouton", locales[1]);
```

La méthode getBundle() recherche en premier une classe qui correspond au nom de base, si elle n'existe pas alors elle recherche un fichier de propriétés. Lorsque le fichier est trouvé, elle retourne un objet PropertyResourceBundle qui contient les paires clé-valeur du fichier

Pour retrouver la traduction d'un texte, il faut utiliser la méthode getString() d'un objet ResourceBundle

Exemple (code Java 1.1) :

```
String valeur = titres.getString(key);
```

Lors du débogage, il peut être utile d'obtenir la liste de paire d'un objet ResourceBundle. La méthode getKeys() retourne un objet Enumeration qui contient toutes les clés de l'objet.

Exemple (code Java 1.1) :

```
ResourceBundle titres =ResourceBundle.getBundle("TitreBouton", locales[1});
Enumeration cles = titres.getKeys();
while (bundleKeys.hasMoreElements()) {
   String cle = (String)cles.nextElement();
   String valeur = titres.getString(cle);
   System.out.println("cle = " + valeur +
      ", " +   "valeur = " + valeur);
}
```

24.3.2. Des exemples de classes utilisant PropertiesResourceBundle

Exemple (code Java 1.1) : Sources de la classe I18nProperties

```
/*
Test d'utilisation de la classe PropertiesResourceBundle
pour internationaliser une application
13/02/99
*/

import java.util.*;

/**
 * Description de la classe I18nProperties
 *
 * @version       0.10 13 fevrier 1999
 * @author        Jean Michel DOUDOUX
 */
public class I18nProperties {
  /**
   * Constructeur de la classe
   */
  public I18nProperties() {

    String texte;
    Locale locale;
    ResourceBundle res;

    System.out.println("Locale par defaut : ");
    locale = Locale.getDefault();
    res = ResourceBundle.getBundle("I18nPropertiesRessources", locale);
    texte = (String)res.getObject("texte_suivant");
    System.out.println("texte_suivant = "+texte);
    texte = (String)res.getObject("texte_precedent");
    System.out.println("texte_precedent = "+texte);
```

```
    System.out.println("\nLocale anglaise : ");
    locale = new Locale("en","");
    res = ResourceBundle.getBundle("I18nPropertiesRessources", locale);
    texte = (String)res.getObject("texte_suivant");
    System.out.println("texte_suivant = "+texte);
    texte = (String)res.getObject("texte_precedent");
    System.out.println("texte_precedent = "+texte);

    System.out.println("\nLocale allemande : "+
       "non définie donc utilisation locale par defaut ");
    locale = Locale.GERMAN;
    res = ResourceBundle.getBundle("I18nPropertiesRessources", locale);
    texte = (String)res.getObject("texte_suivant");
    System.out.println("texte_suivant = "+texte);
    texte = (String)res.getObject("texte_precedent");
    System.out.println("texte_precedent = "+texte);
  }

 /**
   * Pour tester la classe
   *
   * @param      args[]         arguments passes au programme
   */
  public static void main(String[] args) {
    I18nProperties i18nProperties = new I18nProperties();
  }

}
```

Exemple (code Java 1.1) : Contenu du fichier I18nPropertiesRessources.properties

```
texte_suivant=suivant
texte_precedent=Precedent
```

Exemple (code Java 1.1) : Contenu du fichier I18nPropertiesRessources_en.properties

```
texte_suivant=next
texte_precedent=previous
```

Exemple (code Java 1.1) : Contenu du fichier I18nPropertiesRessources_en_US.properties

```
texte_suivant=next
texte_precedent=previous
```

24.3.3. L'utilisation de la classe ListResourceBundle

Il faut créer autant de sous classes de ListResourceBundle que de langues désirées : ceci va générer un fichier .class pour chacune des langues .

Exemple (code Java 1.1) :

```
TitreBouton_fr_FR.class
TitreBouton_en_EN.class
```

Le nom de la classe doit contenir le nom de base plus le code langue et le code pays séparés par un souligné. A l'intérieur de la classe, un tableau à deux dimensions est initialisé avec les paires clé-valeur. Les clés sont des chaines qui doivent être identiques dans toutes les classes des différentes langues. Les valeurs peuvent être des objets de n'importe quel type.

Exemple (code Java 1.1) :

```
import java.util.*;
```

```
public class TitreBouton_fr_FR extends ListResourceBundle {
   public Object[][] getContents() {
      return contents;
   }

   private Object[][] contents = {
      { "texte_suivant", new String(" suivant ")},
      { "Numero", new Integer(4) }
   };
}
```

Il faut définir un objet de type Locale

Il faut créer un objet de type ResourceBundle en appelant la méthode getBundle() de la classe Locale

Exemple (code Java 1.1) :

```
ResourceBundle titres=ResourceBundle.getBundle("TitreBouton", locale);
```

La méthode getBundle() recherche une classe qui commence par TitreBouton et qui est suivie par le code langue et le code pays précisé dans l'objet Locale passé en paramètre

La méthode getObject() permet d'obtenir la valeur de la clé passée en paramètres. Dans ce cas une conversion est nécessaire.

Exemple (code Java 1.1) :

```
Integer valeur = (Integer)titres.getObject("Numero");
```

24.3.4. Des exemples de classes utilisant ListResourceBundle

Exemple (code Java 1.1) : Sources de la classe I18nList

```
/*
Test d'utilisation de la classe ListResourceBundle
pour internationaliser une application
13/02/99
*/

import java.util.*;

/**
 * Description de la classe I18nList
 *
 * @version       0.10 13 fevrier 1999
 * @author        Jean Michel DOUDOUX
 */
public class I18nList {
  /**
   * Constructeur de la classe
   */
  public I18nList() {

    String texte;
    Locale locale;
    ResourceBundle res;

    System.out.println("Locale par defaut : ");
    locale = Locale.getDefault();
    res = ResourceBundle.getBundle("I18nListRessources", locale);
    texte = (String)res.getObject("texte_suivant");
    System.out.println("texte_suivant = "+texte);
    texte = (String)res.getObject("texte_precedent");
```

```
    System.out.println("texte_precedent = "+texte);

    System.out.println("\nLocale anglaise : ");
    locale = new Locale("en","");
    res = ResourceBundle.getBundle("I18nListRessources", locale);
    texte = (String)res.getObject("texte_suivant");
    System.out.println("texte_suivant = "+texte);
    texte = (String)res.getObject("texte_precedent");
    System.out.println("texte_precedent = "+texte);

    System.out.println("\nLocale allemande : "+
       "non définie donc utilisation locale par defaut ");
    locale = Locale.GERMAN;
    res = ResourceBundle.getBundle("I18nListRessources", locale);
    texte = (String)res.getObject("texte_suivant");
    System.out.println("texte_suivant = "+texte);
    texte = (String)res.getObject("texte_precedent");
    System.out.println("texte_precedent = "+texte);
  }

 /**
   * Pour tester la classe
   *
   * @param      args[]         arguments passes au programme
   */
  public static void main(String[] args) {
    I18nList i18nList = new I18nList();
  }

}
```

Exemple (code Java 1.1) : Sources de la classe I18nListRessources

```
/*
test d'utilisation de la classe ListResourceBundle pour
internationaliser une application
13/02/99
*/

import java.util.*;

/**
 * Ressource contenant les traductions françaises
 * langue par defaut de l'application
 *
 * @version       0.10 13 fevrier 1999
 * @author        Jean Michel DOUDOUX
 *
 */

public class I18nListRessources extends ListResourceBundle {
  public Object[][] getContents() {
    return contents;
  }

   //tableau des mots clés et des valeurs

   static final Object[][] contents = {
   {"texte_suivant", "Suivant"},
   {"texte_precedent", "Precedent"},
  };
}
```

Exemple (code Java 1.1) : Sources de la classe I18nListRessources_en

```
/*
test d'utilisation de la classe ListResourceBundle pour
internationaliser une application
13/02/99
*/

import java.util.*;
```

```
/**
 * Ressource contenant les traductions anglaises
 *
 * @version       0.10 13 fevrier 1999
 * @author        Jean Michel DOUDOUX
 *
 */
public class I18nListRessources_en extends ListResourceBundle {
  public Object[][] getContents() {
    return contents;
  }

   //tableau des mots clés et des valeurs

   static final Object[][] contents = {
   {"texte_suivant", "Next"},
   {"texte_precedent", "Previous"},
  };
}
```

Exemple (code Java 1.1) : Sources de la classe I18nListRessources_en_US

```
/*
test d'utilisation de la classe ListResourceBundle pour
internationaliser une application
13/02/99
*/
import java.util.*;
/**
* Ressource contenant les traductions américaines
*
* @version 0.10 13 fevrier 1999
* @author Jean Michel DOUDOUX
*
*/

public class I18nListRessources_en_US extends ListResourceBundle {
public Object[][] getContents() {
return contents;
}

//tableau des mots clés et des valeurs

static final Object[][] contents = {
{"texte_suivant", "Next"},
{"texte_precedent", "Previous"},
};
}
```

24.3.5. La création de sa propre classe fille de ResourceBundle

La troisième solution consiste à créer sa propre sous classe de ResourceBundle et à surcharger la méthode handleGetObject().

Exemple (code Java 1.1) :

```
abstract class MesRessources extends ResourceBundle {

   public Object handleGetObject(String cle)  {
      if(cle.equals(" texte_suivant "))
         return " Suivant " ;
      if(cle.equals(" texte_precedent "))
         return "Precedent " ;
         return null ;
   }
}
```

Attention : la classe ResourceBundle contient deux méthodes abstraites : handleGetObjects() et getKeys(). Si l'une des deux n'est pas définie alors il faut définir la sous classe avec le mot clé abstract.

Il faut créer autant de sous classes que de Locale désiré : il suffit simplement d'ajouter dans le nom de la classe le code langue et le code pays avec éventuellement le code variant.

Exemple (code Java 1.1) : Sources de la classe I18nResource

```
/*
Test d'utilisation d'un sous classement
de la classe ResourceBundle pour
internationaliser une application
13/02/99
*/
import java.util.*;
/**
 * Description de la classe I18nResource
 *
 * @version       0.10 13 fevrier 1999
 * @author        Jean Michel DOUDOUX
 */
public class I18nResource {
  /**
   * Constructeur de la classe
   */
  public I18nResource() {

    String texte;
    Locale locale;
    ResourceBundle res;


    System.out.println("Locale par defaut : ");
    res = ResourceBundle.getBundle("I18nResourceBundle");
    texte = (String)res.getObject("texte_suivant");
    System.out.println("texte_suivant = "+texte);
    texte = (String)res.getObject("texte_precedent");
    System.out.println("texte_precedent = "+texte);


    System.out.println("\nLocale anglaise : ");
    locale = new Locale("en","");
    res = ResourceBundle.getBundle("I18nResourceBundle", locale);
    texte = (String)res.getObject("texte_suivant");
    System.out.println("texte_suivant = "+texte);
    texte = (String)res.getObject("texte_precedent");
    System.out.println("texte_precedent = "+texte);

    System.out.println("\nLocale allemande : "+
       "non définie donc utilisation locale par defaut ");
    locale = Locale.GERMAN;
    res = ResourceBundle.getBundle("I18nResourceBundle", locale);
    texte = (String)res.getObject("texte_suivant");
    System.out.println("texte_suivant = "+texte);
    texte = (String)res.getObject("texte_precedent");
    System.out.println("texte_precedent = "+texte);

  }

 /**
   * Pour tester la classe
   *
   * @param      args[]         arguments passes au programme
   */
  public static void main(String[] args) {
    I18nResource i18nResource = new I18nResource();
  }

}
```

Exemple (code Java 1.1) : Sources de la classe I18nResourceBundle

```
/*
Test d'utilisation de la derivation de la classe ResourceBundle pour
internationaliser une application
13/02/99
*/

import java.util.*;

/**
 * Description de la classe I18nResourceBundle
 * C'est la classe contenant la locale par defaut
 * Contient les traductions de la locale francaise (langue par defaut)
 * Elle herite de ResourceBundle
 *
 * @version       0.10 13 fevrier 1999
 * @author        Jean Michel DOUDOUX
 */
public class I18nResourceBundle extends ResourceBundle {
   protected Vector table;

   public I18nResourceBundle() {
      super();
      table = new Vector();
      table.addElement("texte_suivant");
      table.addElement("texte_precedent");
   }

   public Object handleGetObject(String cle)  {
      if(cle.equals(table.elementAt(0))) return "Suivant" ;
      if(cle.equals(table.elementAt(1))) return "Precedent" ;
      return null ;
   }

   public Enumeration getKeys()  {
      return table.elements();
   }

}
```

Exemple (code Java 1.1) : Sources de la classe I18nResourceBundle_en

```
/*
Test d'utilisation de la derivation de la classe ResourceBundle pour
internationaliser une application
13/02/99
*/

import java.util.*;

/**
 * Description de la classe I18nResourceBundle_en
 * Contient les traductions de la locale anglaise
 * Elle herite de la classe contenant la locale par defaut
 *
 * @version       0.10 13 fevrier 1999
 * @author        Jean Michel DOUDOUX
 */
public class I18nResourceBundle_en extends I18nResourceBundle {
   public Object handleGetObject(String cle)  {
      if(cle.equals(table.elementAt(0))) return "Next" ;
      if(cle.equals(table.elementAt(1))) return "Previous" ;
      return null ;
   }
}
```

Exemple (code Java 1.1) : Sources de la classe I18nResourceBundle_fr_FR

```
/*
Test d'utilisation de la derivation de la classe ResourceBundle pour
internationaliser une application
```

```
13/02/99
*/

import java.util.*;

/**
 * Description de la classe I18nResourceBundle_fr_FR
 * Contient les traductions de la locale francaise
 * Elle herite de la classe contenant la locale par defaut
 *
 * @version       0.10 13 fevrier 1999
 * @author        Jean Michel DOUDOUX
 */
public class I18nResourceBundle_fr_FR extends I18nResourceBundle {

   /**
    *
    * Retourne toujours null car la locale francaise correspond
    * a la locale par defaut
    *
    */
   public Object handleGetObject(String cle)  {
      return null ;
   }
}
```

25. Les composants Java beans

Les java beans sont des composants réutilisables introduits par le JDK 1.1. De nombreuses fonctionnalités de ce JDK lui ont été ajoutées pour développer des caractéristiques de ces composants. Les java beans sont couramment appelés simplement beans.

Les beans sont prévus pour pouvoir interagir avec d'autres beans au point de pouvoir développer une application simplement en assemblant des beans avec un outil graphique dédié. Sun fournit gratuitement un tel outil : le B.D.K. (Bean Development Kit).

Ce chapitre contient plusieurs sections :

- La présentation des Java beans
- Les propriétés.
- Les méthodes
- Les événements
- L'introspection
- Paramétrage du bean (Customization)
- La persistance
- La diffusion sous forme de jar

25.1. La présentation des Java beans

Des composants réutilisables sont des objets autonomes qui doivent pouvoir être facilement assemblés entres eux pour créer un programme.

Microsoft propose la technologie ActiveX pour définir des composants mais celle ci est spécifiquement destinée aux plateformes Windows.

Les Java beans proposés par Sun reposent bien sûre sur java et de fait en possèdent toutes les caractéristiques : indépendance de la plate-forme, taille réduite du composant, ...

La technologie java beans propose de simplifier et faciliter la création et l'utilisation de composants.

Les java beans possèdent plusieurs caractéristiques :

- la persistance : elle permet grâce au mécanisme de sérialisation de sauvegarder l'état d'un bean pour le restaurer ainsi si on assemble plusieurs beans pour former une application, on peut la sauvegarder.
- la communication grâce à des événements qui utilise le modèle des écouteurs introduit par java 1.1
- l'introspection : ce mécanisme permet de découvrir de façon dynamique l'ensemble des éléments qui compose le bean (attributs, méthodes et événements) sans avoir le code source.
- la possibilité de paramétrer le composant : les données du paramétrage sont conservées dans des propriétés.

Ainsi, les beans sont des classes java qui doivent respecter un certains nombre de règles :

- ils doivent posséder un constructeur sans paramètre. Celui ci devra initialiser l'état du bean avec des valeurs par défauts.

* ils peuvent définir des propriétés : celles ci sont identifiées par des méthodes dont le nom et la signature sont normalisés
* ils devraient implémenter l'interface serialisable : ceci est obligatoire pour les beans qui possèdent une partie graphique pour permettre la sauvegarde de leur état
* ils définissent des méthodes utilisables par les composants extérieures : elles doivent être public et prévoir une gestion des accès concurrents
* ils peuvent émettre des événements en gérant une liste d'écouteurs qui s'y abonnent via des méthodes dont les noms sont normalisés

Le type de composants le plus adapté est le composant visuel. D'ailleurs, les composants des classes A.W.T. et Swing pour la création d'interfaces graphiques sont tous des beans. Mais les beans peuvent aussi être des composants non visuels pour prendre en charge les traitements.

25.2. Les propriétés.

Les propriétés contiennent des données qui gèrent l'état du composant : ils peuvent être de type primitif ou être un objet.

Il existe quatre types de propriétés :

* les propriétés simples
* les propriétés indexées (indexed properties)
* les propriétés liées (bound properties)
* les propriétés liées avec contraintes (Constrained properties)

25.2.1. Les propriétés simples

Les propriétés sont des variables d'instance du bean qui possèdent des méthodes particulières pour lire et modifier leur valeur. La normalisation de ces méthodes permet à des outils de déterminer de façon dynamique quelles sont les propriétés du bean. L'accès à ces propriétés doit se faire via ces méthodes. Ainsi la variable qui stocke la valeur de la propriété ne doit pas être déclarée public mais les méthodes d'accès à cette variable doivent bien sûre l'être.

Le nom de la méthode de lecture d'une propriété doit obligatoirement commencer par « get » suivi par le nom de la propriété dont la première lettre doit être une majuscule. Une telle méthode est souvent appelée « getter » ou « accesseur » de la propriété. La valeur retournée par cette méthode doit être du type de la propriété.

Exemple (code Java 1.1) :

```
private int longueur;
public int getLongueur () {
  return longueur;
}
```

Pour les propriétés booléennes, une autre convention peut être utilisée : la méthode peut commencer par «is» au lieu de « get ». Dans ce cas, la valeur de retour est obligatoirement de type boolean.

Le nom de la méthode permettant la modification d'une propriété doit obligatoirement commencer par « set » suivi par le nom de la propriété dont la première lettre doit être une majuscule. Une telle méthode est souvent appelée « setter ». Elle ne retourne aucune valeur et doit avoir en paramètre une variable du type de la propriété qui contiendra sa nouvelle valeur. Elle devra assurer la mise à jour de la valeur de la propriété en effectuant éventuellement des contrôles et/ou des traitements (par exemple le rafraîchissement pour un bean visuel dont la propriété affecte l'affichage).

Exemple (code Java 1.1) :

```
private int longueur ;
public void setLongueur (int longueur) {
  this.longueur = longueur;
}
```

Une propriété peut n'avoir qu'un getter et pas de setter : dans ce cas, la propriété n'est utilisable qu'en lecture seule.

Le nom de la variable d'instance qui contient la valeur de la propriété n'est pas obligatoirement le même que le nom de la propriété

Il est préférable d'assurer une gestion des accès concurrents dans ces méthodes de lecture et de mise à jour des propriétés par exemple en déclarant ces méthodes synchronized.

Les méthodes du beans peuvent directement manipuler en lecture et écriture la variable d'instance qui stocke la valeur de la propriété, mais il est préférable d'utiliser le getter et le setter.

25.2.2. Les propriétés indexées (indexed properties)

Ce sont des propriétés qui possèdent plusieurs valeurs stockées dans un tableau.

Pour ces propriétés, il faut aussi définir des méthodes « get » et « set » dont il convient d'ajouter un paramètre de type int représentant l'index de l'élément du tableau.

Exemple (code Java 1.1) :

```
private float[] notes = new float[5];
public float getNotes (int i ) {
  return notes[i];
}
public void setNotes (int i ; float notes) {
  this.notes[i] = notes;
}
```

Il est aussi possible de définir des méthodes « get » et « set » permettant de mettre à jour tout le tableau.

Exemple (code Java 1.1) :

```
private float[] notes = new float[5] ;
public float[] getNotes () {
  return notes;
}
public void setNotes (float[] notes) {
  this.notes = notes;
}
```

25.2.3. Les propriétés liées (Bound properties)

Il est possible d'informer d'autres composants du changement de la valeur d'une propriété d'un bean. Les java beans peuvent mettre en place un mécanisme qui permet pour une propriété d'enregistrer des composants qui seront informés du changement de la valeur de la propriété.

Ce mécanisme peut être mis en place grâce à un objet de la classe PropertyChangeSupport qui permet de simplifier la gestion de la liste des écouteurs et de les informer des changements de valeur d'une propriété. Cette classe définit les méthodes addPropertyChangeListener() pour enregistrer un composant désirant être informé du changement de la valeur de la propriété et removePropertyChangeListener() pour supprimer un composant de la liste.

La méthode firePropertyChange() permet d'informer tous les composants enregistrés du changement de la valeur de la propriété.

Le plus simple est que le bean hérite de cette classe si possible car les méthodes addPropertyChangeListener() et

removePropertyChangeListener() seront directement héritées.

Si ce n'est pas possible, il est obligatoire de définir les méthodes addPropertyChangeListener() et removePropertyChangeListener() dans le bean qui appelleront les méthodes correspondantes de l'objet PropertyChangeSupport.

Exemple (code Java 1.1) :

```
import java.io.Serializable;
import java.beans.*;
public class MonBean03 implements Serializable {
  protected int valeur;

  PropertyChangeSupport changeSupport;

  public MonBean03(){
    valeur = 0;

    changeSupport = new PropertyChangeSupport(this);
  }

  public synchronized void setValeur(int val) {
    int oldValeur = valeur;
    valeur = val;

    changeSupport.firePropertyChange("valeur",oldValeur,valeur);
  }
  public synchronized int getValeur() {
    return valeur;
  }
  public synchronized void addPropertyChangeListener(PropertyChangeListener listener) {
    changeSupport.addPropertyChangeListener(listener);
  }

  public synchronized void removePropertyChangeListener(PropertyChangeListener listener) {
    changeSupport.removePropertyChangeListener(listener);
  }

}
```

Les composants qui désirent être enregistrés doivent obligatoirement implémenter l'interface PropertyChangeListener et définir la méthode propertyChange() déclarée par cette interface.

La méthode propertyChange() reçoit en paramètre un objet de type PropertyChangeEvent qui représente l'événement. Cette méthode de tous les objets enregistrés est appelée. Le paramètre de type PropertyChangeEvent contient plusieurs informations :

- l'objet source : le bean dont la valeur d'une propriété a changé
- le nom de la propriété sous forme de chaîne de caractères
- l'ancienne valeur sous forme d'un objet de type Object
- la nouvelle valeur sous forme d'un objet de type Object

Pour les traitements, il est souvent nécessaire d'utiliser un cast pour transmettre ou utiliser les objets qui représentent l'ancienne et la nouvelle valeur.

Méthode	Rôle
public Object getSource()	retourne l'objet source
public Object getNewValue()	retourne la nouvelle valeur de la propriété
public Object getOldValue()	retourne l'ancienne valeur de la propriété
public String getPropertyName	retourne le nom de la propriété modifiée

Exemple (code Java 1.1) : un programme qui créé le bean et lui associe un écouteur

```
import java.beans.*;
import java.util.*;
public class TestMonBean03 {
  public static void main(String[] args) {
    new TestMonBean03();
  }

  public TestMonBean03() {
    MonBean03 monBean = new MonBean03();

    monBean.addPropertyChangeListener( new PropertyChangeListener() {
      public void propertyChange(PropertyChangeEvent event) {
        System.out.println("propertyChange : valeur = "+ event.getNewValue());
      }
    } );

    System.out.println("valeur = " + monBean.getValeur());
    monBean.setValeur(10);
    System.out.println("valeur = " + monBean.getValeur());

  }
}
```

Résultat :

```
C:\tutorial\sources exemples>java TestMonBean03
valeur = 0
propertyChange : valeur = 10
valeur = 10
```

Pour supprimer un écouteur de la liste du bean, il suffit d'appeler la méthode removePropertyChangeListener() en lui passant en paramètre un référence sur l'écouteur.

25.2.4. Les propriétés liées avec contraintes (Constrained properties)

Ces propriétés permettent à un ou plusieurs composants de mettre un veto sur la modification de la valeur de la propriété.

Comme pour les propriétés liées, le bean doit gérer une liste de composants « écouteurs » qui souhaitent être informé d'un changement possible de la valeur de la propriété. Si un composant désire s'opposer à ce changement de valeur, il lève une exception pour en informer le bean.

Les écouteurs doivent implémenter l'interface VetoableChangeListener qui définit la méthode vetoableChange().

Avant le changement de la valeur, le bean appelle cette méthode vetoableChange() de tous les écouteurs enregistrés. Elle possède en paramètre un objet de type PropertyChangeEvent qui contient : le bean, le nom de la propriété, l'ancienne valeur et la nouvelle valeur.

Si un écouteur veut s'opposer à la mise à jour de la valeur, il lève une exception de type java.beans.PropertyVetoException. Dans ce cas, le bean ne change pas la valeur de la propriété : ces traitements sont à la charge du programmeur avec notamment la gestion de la capture et du traitement de l'exception dans un bloc try/catch.

La classe VetoableChangeSupport permet de simplifier la gestion de la liste des écouteurs et de les informer du futur changement de valeur d'une propriété. Son utilisation est similaire à celle de la classe PropertyChangeSupport.

Pour ces propriétés, pour que les traitements soient complets il faut implémenter le code pour gérer et traiter les écouteurs qui souhaitent connaître les changements de valeur effectifs de la propriété (voir les propriétés liées).

Exemple (code Java 1.1) :

```
import java.io.Serializable;
import java.beans.*;
public class MonBean04 implements Serializable {
  protected int oldValeur;
```

```
  protected int valeur;

  PropertyChangeSupport changeSupport;
  VetoableChangeSupport vetoableSupport;

  public MonBean04(){
    valeur = 0;
    oldValeur = 0;

    changeSupport = new PropertyChangeSupport(this);
    vetoableSupport = new VetoableChangeSupport(this);
  }


  public synchronized void setValeur(int val) {
    oldValeur = valeur;
    valeur = val;

    try {
      vetoableSupport.fireVetoableChange("valeur",new Integer(oldValeur),
              new Integer(valeur));
    } catch(PropertyVetoException e) {
      System.out.println("MonBean, un veto est emis :  "+e.getMessage());
      valeur = oldValeur;
    }
    if ( valeur != oldValeur ) {
      changeSupport.firePropertyChange("valeur",oldValeur,valeur);
    }

  }
  public synchronized int getValeur() {
    return valeur;
  }
  public synchronized void addPropertyChangeListener(PropertyChangeListener listener) {
    changeSupport.addPropertyChangeListener(listener);
  }

  public synchronized void removePropertyChangeListener(PropertyChangeListener listener) {
    changeSupport.removePropertyChangeListener(listener);
  }
  public synchronized void addVetoableChangeListener(VetoableChangeListener listener) {
    vetoableSupport.addVetoableChangeListener(listener);
  }
  public synchronized void removeVetoableChangeListener(VetoableChangeListener listener) {
    vetoableSupport.removeVetoableChangeListener(listener);
  }

}
```

Exemple (code Java 1.1) : un programme qui teste le bean. Il émet un veto si la nouvelle valeur de la propriété est supérieure à 100.

```
import java.beans.*;
import java.util.*;
public class TestMonBean04 {
  public static void main(String[] args) {
    new TestMonBean04();
  }

  public TestMonBean04() {
    MonBean04 monBean = new MonBean04();

    monBean.addPropertyChangeListener( new PropertyChangeListener() {
      public void propertyChange(PropertyChangeEvent event) {
        System.out.println("propertyChange : valeur = "+ event.getNewValue());
      }
    } );

    monBean.addVetoableChangeListener( new VetoableChangeListener() {

      public void vetoableChange(PropertyChangeEvent event) throws PropertyVetoException {
        System.out.println("vetoableChange : valeur = " + event.getNewValue());
        if( ((Integer)event.getNewValue()).intValue() > 100 )
```

```
            throw new PropertyVetoException("valeur superieur a 100",event);
          }
      } );

      System.out.println("valeur = " + monBean.getValeur());
      monBean.setValeur(10);
      System.out.println("valeur = " + monBean.getValeur());
      monBean.setValeur(200);
      System.out.println("valeur = " + monBean.getValeur());

    }
}
```

Exemple (code Java 1.1) :

```
C:\tutorial\sources exemples>java TestMonBean04
valeur = 0
vetoableChange : valeur = 10
propertyChange : valeur = 10
valeur = 10
vetoableChange : valeur = 200
vetoableChange : valeur = 10
MonBean, un veto est emis :  valeur superieur a 100
valeur = 10
```

25.3. Les méthodes

Toutes les méthodes publiques sont visibles de l'extérieur et peuvent donc être appelées.

25.4. Les événements

Les beans utilisent les événements définis dans le modèle par délégation introduit par le J.D.K. 1.1 pour dialoguer. Par respect de ce modèle, le bean est la source et les autres composants qui souhaitent être informés sont nommés « Listeners » ou « écouteurs » et doivent s'enregistrer auprès du bean qui maintient la liste des composants enregistrés.

Il est nécessaire de définir les méthodes qui vont permettre de gérer la liste des écouteurs désirant recevoir l'événement. Il faut définir deux méthodes :

- public void addXXXListener(XXXListener li) pour enregistrer l'écouteur li
- public void removeXXXListener (XXXListener li) pour enlever l'écouteur li de la liste

L'objet de type XXXListener doit obligatoirement implémenter l'interface java.util.EventListener et son nom doit terminer par « Listener ».

Les événements peuvent être mono ou multi écouteurs.

Pour les événements mono écouteurs, la méthode addXXXListener() doit indiquer dans sa signature qu'elle est susceptible de lever l'exception java.util.TooManyListenersException si un écouteur tente de s'enregistrer et qu'il y en a déjà un présent.

25.5. L'introspection

L'introspection est un mécanisme qui permet de déterminer de façon dynamique les caractéristiques d'une classe et donc d'un bean. Les caractéristiques les plus importantes sont les propriétés, les méthodes et les événements. Le principe de l'introspection permet à Sun d'éviter de rajouter des éléments au langage pour définir ces caractéristiques.

L'API JavaBean définit la classe java.beans.Introspector qui facilite et standardise la recherche des propriétés, méthodes et événements du bean. Cette classe possède des méthodes pour analyser le bean et retourner un objet de type BeanInfo contenant les informations trouvées.

La classe Introspector utilise deux techniques pour retrouver ces informations :

1. un objet de type BeanInfo, si il y en a un défini par les développeurs du bean
2. les mécanismes fournis par l'API reflexion pour extraire les entités qui respectent leur modèle (design pattern) respectif.

Il est donc possible de définir un objet BeanInfo qui sera directement utilisé par la classe Introspector. Cette définition est utile si le bean ne respecte pas certains modèles (designs patterns) ou si certaines entités héritées ne doivent pas être utilisables. Dans ce cas, le nom de cette classe doit obligatoirement respecter le modèle XXXBeanInfo ou XXX est le nom du bean correspondant. La classe Introspector recherche une classe respectant ce modèle.

Si une classe BeanInfo pour un bean est définie, une classe qui hérite du bean n'est pas obligée de définir un classe BeanInfo. Dans ce cas, la classe Introspector utilise les informations du BeanInfo de la classe mère et ajoute les informations retournées par l'API reflection sur le bean.

Sans classe BeanInfo associée au bean, les méthodes de la classe Introspector utilisent les techniques d'introspection pour analyser le bean.

25.5.1. Les modèles (designs patterns)

Si la classe Introspector utilise l'API reflection pour déterminer les informations sur le bean et utilise en même temps un ensemble de modèles sur chacunes des entités propriétés, méthodes et événements.

Pour déterminer les propriétés, la classe Introspector recherche les méthodes getXxx(), setXxx() et isXxx() ou Xxx représente le nom de la propriété dont la première lettre est en majuscule. La première lettre du nom de la propriété est remise en minuscule sauf si les deux premières lettres de la propriété sont en majuscules.

Pour déterminer les méthodes, la classe Introspector rechercher toutes les méthodes publiques.

Pour déterminer les événements, la classe Introspector recherche les méthodes addXxxListener() et removeXxxListener(). Si les deux sont présentes, elle en déduit que l'événement xxx est défini dans le bean. Comme pour les propriétés, la première lettre du nom de l'événement est mise en minuscule.

25.5.2. La classe BeanInfo

La classe BeanInfo contient des informations sur un bean et possède plusieurs méthodes pour les obtenir.

La méthode getBeanInfo() prend en paramètre un objet de type Class qui représente la classe du bean et elle renvoie des informations sur la classe et toutes ses classes mères.

Une version surchargée de la méthode accepte deux objets de type Class : le premier représente le bean et le deuxième représente une classe appartenant à la hiérarchie du bean. Dans ce cas, la recherche d'informations d'arrêtera juste avant d'arriver à la classe précisée en deuxième argument.

Exemple : obtenir des informations sur le bean uniquement (sans informations sur ces supers classes)

Exemple (code Java 1.1) :

```
Class monBeanClasse = Class.forName("monBean");
BeanInfo bi = Introspector.getBeanInfo(monBeanClasse, monBeanClasse.getSuperclass());
```

La méthode getBeanDescriptor() permet d'obtenir des informations générales sur le bean en renvoyant un objet de type BeanDescriptor()

La méthode getPropertyDescriptors() permet d'obtenir un tableau d'objets de type PropertyDescriptor qui contient les caractéristiques d'une propriété. Plusieurs méthodes permettent d'obtenir ces informations.

Exemple (code Java 1.1) :

```
    PropertyDescriptor[] propertyDescriptor = bi.getPropertyDescriptors();
    for (int i=0; i<propertyDescriptor.length; i++) {
      System.out.println(" Nom propriete    : " +
                        propertyDescriptor[i].getName());
      System.out.println(" Type propriete   : "
                  + propertyDescriptor[i].getPropertyType());
      System.out.println(" Getter propriete : "
                    + propertyDescriptor[i].getReadMethod());
      System.out.println(" Setter propriete : "
                   + propertyDescriptor[i].getWriteMethod());
    }
```

La méthode getMethodDescriptors() permet d'obtenir un tableau d'objets de type MethodDescriptor qui contient les caractéristiques d'une méthode. Plusieurs méthodes permettent d'obtenir ces informations.

Exemple (code Java 1.1) :

```
    MethodDescriptor[] methodDescriptor;
    unMethodDescriptor = bi.getMethodDescriptors();
    for (int i=0; i < unMethodDescriptor.length; i++) {
      System.out.println(" Methode : "+unMethodDescriptor[i].getName());
    }
```

La méthode getEventSetDescriptors() permet d'obtenir un tableau d'objets de type EventSetDescriptor qui contient les caractéristiques d'un événement. Plusieurs méthodes permettent d'obtenir ces informations.

Exemple (code Java 1.1) :

```
    EventSetDescriptor[] unEventSetDescriptor = bi.getEventSetDescriptors();
    for (int i = 0; i < unEventSetDescriptor.length; i++) {
      System.out.println(" Nom evt             : "
                      + unEventSetDescriptor[i].getName());
      System.out.println(" Methode add evt     : " +
             unEventSetDescriptor[i].getAddListenerMethod());
      System.out.println(" Methode remove evt  : " +
          unEventSetDescriptor[i].getRemoveListenerMethod());
      unMethodDescriptor = unEventSetDescriptor[i].getListenerMethodDescriptors();
      for (int j = 0; j < unMethodDescriptor.length; j++) {
          System.out.println(" Event Type: " + unMethodDescriptor[j].getName());
      }
    }
```

Exemple (code Java 1.1) :

```
import java.io.*;
import java.beans.*;
import java.lang.reflect.*;
public class BeanIntrospection {
  static String nomBean;
  public static void main(String args[]) throws Exception {
    nomBean = args[0];
    new BeanIntrospection();
  }

  public BeanIntrospection() throws Exception {
    Class monBeanClasse = Class.forName(nomBean);
    MethodDescriptor[] unMethodDescriptor;

    BeanInfo bi = Introspector.getBeanInfo(monBeanClasse, monBeanClasse.getSuperclass());
    BeanDescriptor unBeanDescriptor = bi.getBeanDescriptor();
    System.out.println("Nom du bean    : " + unBeanDescriptor.getName());
    System.out.println("Classe du bean : " + unBeanDescriptor.getBeanClass());
```

```
    System.out.println("");

    PropertyDescriptor[] propertyDescriptor = bi.getPropertyDescriptors();
    for (int i=0; i<propertyDescriptor.length; i++) {
      System.out.println(" Nom propriete    : " +
                          propertyDescriptor[i].getName());
      System.out.println(" Type propriete   : "
                + propertyDescriptor[i].getPropertyType());
      System.out.println(" Getter propriete : "
                  + propertyDescriptor[i].getReadMethod());
      System.out.println(" Setter propriete : "
                 + propertyDescriptor[i].getWriteMethod());
    }
    System.out.println("");
    unMethodDescriptor = bi.getMethodDescriptors();
    for (int i=0; i < unMethodDescriptor.length; i++) {
      System.out.println(" Methode : "+unMethodDescriptor[i].getName());
    }
    System.out.println("");
    EventSetDescriptor[] unEventSetDescriptor = bi.getEventSetDescriptors();
    for (int i = 0; i < unEventSetDescriptor.length; i++) {
      System.out.println(" Nom evt            : "
                        + unEventSetDescriptor[i].getName());
      System.out.println(" Methode add evt     : " +
             unEventSetDescriptor[i].getAddListenerMethod());
      System.out.println(" Methode remove evt  : " +
          unEventSetDescriptor[i].getRemoveListenerMethod());
      unMethodDescriptor = unEventSetDescriptor[i].getListenerMethodDescriptors();
      for (int j = 0; j < unMethodDescriptor.length; j++) {
          System.out.println(" Event Type: " + unMethodDescriptor[j].getName());
      }
    }
    System.out.println("");
  }
}
```

25.6. Paramétrage du bean (Customization)

Il est possible de développer un éditeur de propriétés spécifique pour permettre de personnaliser la modification des paramètres du bean.

La suite de cette section sera développée dans une version future de ce document

25.7. La persistance

Les propriétés du bean doivent pouvoir être sauvés pour être restitués ultérieurement. Le mécanisme utilisé est la sérialisation. Pour permettre d'utiliser ce mécanisme, le bean doit implémenter l'interface Serializable

25.8. La diffusion sous forme de jar

Pour diffuser un bean sous forme de jar, il faut définir un fichier manifest.

Ce fichier doit obligatoirement contenir un attribut Name: qui contient le nom complet de la classe (incluant le package) et un attribut Java-Bean: valorisé à True.

Exemple de fichier manifest pour un bean :

```
Name: MonBean.class
Java-Bean: True
```

La suite de cette section sera développée dans une version future de ce document

26. Logging

Le logging consiste à ajouter des traitements dans les applications pour permettre l'émission et le stockage de messages suite à des événements.

Le logging est utile pour tous les types d'applications en permettant notamment par exemple de conserver une trace des exceptions qui sont levées dans l'application et des différents événements anormaux ou normaux liés à l'exécution de l'application.

Le logging permet de gérer des messages émis par une application durant son exécution et de permettre leur exploitation immédiate ou à postériori. Ces messages sont d'ailleurs très utiles lors de la mise en point d'une application ou lors de son exploitation pour comprendre son fonctionnement ou résoudre une anomalie.

Ce chapitre contient plusieurs sections :

- La présentation du logging
- Log4j
- L'API logging
- Jakarta Commons Logging (JCL)
- D'autres API de logging

26.1. La présentation du logging

Le logging est une activité technique utile et nécessaire dans une application pour :

- Déboguer : pratique lorsque la mise en oeuvre d'un débogueur n'est pas facile.
- Obtenir des traces d'exécution (démarrage/arrêt, informations, avertissements, erreurs d'exécution, ...
- Faciliter la recherche d'une source d'anomalie (stacktrace, ...)
- Comprendre ou vérifier le flux des traitements exécutés : traces des entrées/sorties dans les méthodes, affichage de la pile d'appels, ...
- ...

L'importance du logging croit avec la taille et la complexité de l'application qui l'utilise.

Une API de logging fait généralement intervenir trois composants principaux :

- Logger : invoqué pour émettre via le framework un message généralement avec un niveau de gravité associé
- Formatter : utilisé pour formater le contenu du message
- Appender : utilisé pour envoyer le message à une cible de stockage (console, fichier, base de données, email, ...)

Le logging doit faire partie intégrante des fonctionnalités d'une application. Bien sûre le niveau de gravité des messages n'est pas le même en développement et en production mais le code de l'application doit rester le même. Seule la configuration du logging doit changer dans les différents environnements.

Généralement la configuration peut être externalisée dans un fichier ce qui rend l'utilisation de l'API plus souple et flexible.

La modification de la configuration du logging en cours d'exécution de l'application (soit dynamiquement soit par rechargement de la configuration) est importante pour permettre d'avoir couramment un niveau de log acceptable et d'avoir au besoin un niveau de log plus fin sans avoir à relancer l'application.

Les API de logging ont plusieurs inconvénients :

- Il faut définir avec précision les messages à ajouter dans les journaux et la pertinence des informations qu'ils contiennent
- Il faut définir avec précision le niveau de gravité des messages
- L'utilisation d'une API de logging peut dégrader les performances d'une application

Le logging est particulièrement important dans une application notamment côté serveur mais une utilisation à outrance ou une mauvaise utilisation de cette fonctionnalité peut dégrader les performances générales de l'application.

Les frameworks de logging sont conçus pour limiter la consommation en ressources nécessaires à leur mise en oeuvre mais elle existe tout même et croit naturellement avec le nombre de messages émis.

L'utilisation d'une API de Logging implique donc une surcharge de consommation de ressources (CPU, mémoire, ...) mais elle se justifie par l'apport des informations fournies en cas de problème sous réserve que ces informations aient été judicieusement choisies.

26.1.1. Des recommandations lors de la mise en oeuvre

Voici quelques règles pour une bonne mise en oeuvre du logging :

- Chaque message doit contenir la date/heure d'émission et la classe émettrice
- Ne jamais utiliser de System.out pour afficher des messages mais utiliser une API de Logging
- Ne jamais utiliser la méthode printStackTrace() de la classe Exception pour afficher des messages mais utiliser une API de Logging
- Ne pas émettre de messages qui seront émis très fréquemment (par exemple dans une boucle avec un nombre important d'itérations ou dans une méthode fréquemment invoquée, ...)
- Utiliser le niveau de gravité adéquate avec le message

Pour des traces d'exécution, il est pratique d'émettre un message en début d'une méthode qui affiche les paramètres en entrée et un message à la fin de la méthode avec la valeur de retour

Il est fortement recommandé d'utiliser une API de logging plutôt que d'utiliser la méthode System.out.println() pour plusieurs raisons :

- Une API de logging permet un contrôle sur le format des messages en proposant un format standard pouvant inclure des données telles que la date/heure, la classe, le thread, ...
- Une API de logging permet de gérer différentes cibles de stockage des messages
- Une API de logging permet de modifier à l'exécution le niveau de gravité des messages pris en compte

Sur des applications utilisées par plusieurs utilisateurs, par exemple une application web, il peut très utile de faire figurer dans le message une identité sur le responsable de l'action (par exemple, l'adresse IP d'une requête http).

26.1.2. Les différents frameworks

De nombreux frameworks existent pour mettre en oeuvre le logging dont :

- Log4j
- Java Logging
- Jlog
- Protomatter
- SLF4J
- LogBack

Log4j du groupe Apache Jakarta est sûrement l'API la plus répandue et la plus populaire.

Les qualités de Log4j notamment sa simplicité de mise en oeuvre, ses fonctionnalités, sa fiabilité et son évolutivité lui permettent d'être le standard de facto pour le logging.

Depuis la version 1.4 du JDK, Java intègre une API de logging qui est le standard officiel pour le logging mais qui est légèrement moins riche en fonctionnalité que Log4J mais possède l'avantage d'être fournie dans les API de base.

Afin de faciliter l'utilisation du logging, le groupe Jakarta a développé un wrapper nommé JCL (JakartaCommon Logging) qui permet d'utiliser de façon transparente Log4j ou l'API Logging du JDK en utilisant le tronc commun de ces deux API.

26.2. Log4j

Log4j est un projet open source distribué sous la licence Apache Software initialement créé par Ceki Gülcü et maintenu par le groupe Jakarta. Cette API permet aux développeurs d'utiliser et de paramétrer un système de gestion de journaux (logs). Il est possible de fournir les paramètres dans un fichier de configuration ce qui rend sa configuration facile et souple. Log4j est compatible avec le JDK 1.1. et supérieur.

Log4j gère plusieurs niveaux de gravités et les messages peuvent être envoyés dans plusieurs flux : un fichier sur disque, le journal des événements de Windows, une connexion TCP/IP, une base de données, un message JMS, etc ...

Log4j utilise trois composants principaux pour assurer l'envoi de messages selon un certain niveau de gravité et contrôler à l'exécution le format et la ou les cibles de destination des messages :

- Category/Logger : ces classes permettent de gérer les messages associés à un niveau de gravité
- Appenders : ils représentent les flux qui vont recevoir les messages de log
- Layouts : ils permettent de formater le contenu des messages de log

Ces trois types de composants sont utilisés ensemble pour émettre des messages vers différentes cibles de stockage.

Ceci permet au framework de déterminer les messages qui doivent être loggués, la façon dont ils seront formatés et vers quelle cible les messages seront envoyés.

La popularité de Log4J est largement liée à sa facilité d'utilisation, ses nombreuses fonctionnalités extensibles et sa fiabilité. Comme le logging n'est jamais une fonctionnalité principale d'une application, Log4j se veut facile à mettre en oeuvre.

Les principales caractéristiques de Log4j sont :

- Utilisation d'une hiérarchie de logger basée sur leur nom
- Support en standard de plusieurs niveaux de gravité
- Configuration externalisable dans un fichier au format .properties ou XML
- Thread-safe
- Optimisé pour réduire les temps de traitements
- Prise en charge des exceptions associables aux messages
- Support de nombreuses cibles de destination des messages
- Extensible

Un autre avantage de log4J est de pouvoir être utilisé avec toutes les versions du JDK depuis la 1.1.

L'externalisation de la configuration de Log4j dans un fichier externe permet de modifier la configuration des traitements de logging sans avoir à modifier le code source de l'application.

La hiérarchie des loggers permet un contrôle très fin de la granularité des messages ce qui permet de réduire le volume de données des logs.

Log4j propose en standard plusieurs destinations de stockage des messages : fichiers, gestion d'événements Windows, Syslog Unix, base de données, email, message JMS, ...

L'API Log4j est regroupée dans plusieurs packages :

Package	Rôle
org.apache.log4j	Contient les principales classes et interfaces
org.apache.log4j.spi	System Programming Interface pour étendre Log4j
org.apache.log4j.chainsaw	Application Swing Chainsaw pour visualiser les logs formatée par un XMLLayout ou émise par un SocketAppender
org.apache.log4j.config	Classes pour la gestion des propriétés des composants
org.apache.log4j.helpers	Utilitaires
org.apache.log4j.jdbc	Classes pour stocker les messages dans une base de données
org.apache.log4j.jmx	Classes pour permettre la configuration de Log4j via JMX
org.apache.log4j.lf5	Application Swing Log Force 5 pour visualiser les logs
org.apache.log4j.net	Classes pour envoyer les messages au travers le réseau (JMS, SMTP, Sockets, ...)
org.apache.log4j.nt	Classes pour envoyer les messages dans le système de gestion des événements de Windows
org.apache.log4j.or	Utilitaires pour formater des objets
org.apache.log4j.performance	Classes de tests des performances
org.apache.log4j.xml	Classes pour permettre la configuration de Log4j avec un fichier XML
org.apache.log4j.varia	Classes diverses

Le site officiel de Log4j est à l'url : http://logging.apache.org/log4j/

Log4j est disponible dans trois versions majeures :

- 1.2 : c'est la version stable courante
- 1.3 : cette version est abandonnée
- 2.0 : c'est la future version en cours de développement

26.2.1. Les premiers pas

Cette section fournit des informations et un premier exemple pour la mise en oeuvre de Tomcat.

26.2.1.1. L'installation

Il faut télécharger le fichier apache-log4j-1.2.xx.zip à l'url http://logging.apache.org/log4j/1.2/download.html

Il suffit ensuite de décompresser l'archive dans un répertoire du système. L'archive contient entre autre les sources, la documentation, des exemples et la bibliothèque log4j-1.2.x.jar.

26.2.1.2. Les principes de mise en oeuvre

Pour utiliser Log4j, il suffit d'ajouter le fichier log4j-1.2.x.jar dans le classpath de l'application.

Il faut définir un fichier de configuration : configuration des loggers, définition des appenders, association des appenders aux loggers avec un layout.

Dans le code source des classes, il faut :

- obtenir une instance du logger relatif à la classe
- utiliser l'API pour émettre un message associé à un niveau de gravité

26.2.1.3. Un exemple de mise en oeuvre

Cette section va mettre en oeuvre Log4j dans un exemple très simple.

Il faut créer un fichier log4j.properties stocké dans le classpath de l'application : ce fichier contient la configuration de Log4j pour l'application.

Exemple :

```
log4j.rootLogger=DEBUG, stdout
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d [%-5p] (%F:%M:%L) %m%n
```

Cette configuration définit le niveau de gravité DEBUG pour le logger racine et lui associe un logger nommé arbitrairement stdout. Par héritage, tous les loggers de l'application vont hériter de cette configuration.

L'appender nommé stdout est de type ConsoleAppender : il envoie les messages sur la console standard.

Un layout personnalisé est associé à l'appender nommé stdout pour formater les messages. Chaque séquence commençant par le caractère % sera remplacé dynamiquement par sa valeur correspondante. Par exemple : %d correspond à la date/heure, %p au niveau de gravité, %m le message, %n un retour chariot, ...

Pour mettre en oeuvre l'API dans le code source, il faut tout d'abord obtenir une instance du logger à utiliser en utilisant la méthode getLogger() de la classe Logger.

Chaque message est émis en utilisant la méthode correspondant au niveau de gravité choisi de la classe Logger.

Exemple :

```
package com.jmdoudoux.test.log4j;

import org.apache.log4j.Logger;

public class TestLog4j1 {

  private static Logger logger = Logger.getLogger(TestLog4j1.class);

  public static void main(String[] args) {
    logger.debug("msg de debogage");
    logger.info("msg d'information");
    logger.warn("msg d'avertissement");
    logger.error("msg d'erreur");
    logger.fatal("msg d'erreur fatale");
  }
}
```

L'exécution de cette classe permet d'afficher sur la console les différents messages

Résultat :

```
2008-06-08 10:16:21,546 [DEBUG] (TestLog4j1.java:main:13) msg de debogage
2008-06-08 10:16:21,546 [INFO ] (TestLog4j1.java:main:14) msg d'information
2008-06-08 10:16:21,546 [WARN ] (TestLog4j1.java:main:15) msg d'avertissement
2008-06-08 10:16:21,546 [ERROR] (TestLog4j1.java:main:16) msg d'erreur
2008-06-08 10:16:21,546 [FATAL] (TestLog4j1.java:main:17) msg d'erreur fatale
```

Une simple modification du fichier de configuration permet de changer le niveau de gravité des messages pris en compte. Par exemple en remplaçant DEBUG par ERROR

La réexécution de la classe qui n'a pas été modifiée et donc pas recompilée permet d'afficher sur la console uniquement les messages dont la gravité est supérieure ou égale à ERROR.

Résultat :

```
2008-06-08 10:18:47,530 [ERROR] (TestLog4j1.java:main:13) msg d'erreur
2008-06-08 10:18:47,530 [FATAL] (TestLog4j1.java:main:14) msg d'erreur fatale
```

26.2.2. La gestion des logs avec les versions antérieures à la 1.2

Les versions antérieures à la 1.2 de Log4J utilisent la classe Category pour gérer les messages et la classe Priority pour encapsuler les niveaux de gravité.

26.2.2.1. Les niveaux de gravités : la classe Priority

Log4j gère des priorités pour permettre à une instance de la classe Category de déterminer si le message sera envoyé dans la log ou non. Il existe cinq priorités qui possèdent un ordre hiérarchique croissant :

- DEBUG
- INFO
- WARN
- ERROR
- FATAL

La classe org.apache.log4j.Priority encapsule ces priorités.

Chaque Category est associée à une priorité qui peut être changée dynamiquement. La catégorie détermine si un message doit être envoyé dans la log en comparant sa priorité avec la priorité du message. Si celle-ci est supérieure ou égale à la priorité de la Category, alors le message est envoyé vers la cible de destination de la log.

La méthode setPropriety() de la classe Category permet de préciser la priorité.

Si aucune priorité n'est donnée à une catégorie, elle "hérite" de la priorité de la première catégorie en remontant dans la hiérarchie dont la priorité est renseignée.

Exemple : soit trois catégories
root associée à la priorité INFO
categorie1 nommée "org" sans priorité particulière
categorie2 nommée "org.moi" associée à la priorité ERROR
categorie3 nommée "org.moi.projet" sans priorité particulière

Une demande d'émission de message avec la priorité DEBUG sur categorie1 n'est pas traitée car la priorité INFO héritée est supérieure à DEBUG.
Une demande avec la priorité WARN sur categorie1 est traitée car la priorité INFO héritée est inférieure à WARN .
Une demande avec la priorité DEBUG sur categorie3 n'est pas traitée car la priorité ERROR héritée est supérieure à

DEBUG.
Une demande avec la priorité FATAL sur categorie3 est traitée car la priorité ERROR héritée est inférieure à FATAL.
En fait dans l'exemple, aucune demande avec la priorité DEBUG ne sera traitée.

Au niveau applicatif, il est possible d'interdire le traitement d'une priorité et de celle inférieure en utilisant le code suivant : Category.getDefaultHierarchy().disable(). Il faut fournir la priorité à la méthode disable().

Il est possible d'annuler ce traitement dynamiquement en positionnement la propriété système log4j.disableOverride.

26.2.2.2. La classe Category

La classe org.apache.log4j.Category détermine si un message doit être envoyé dans la ou les logs qui lui sont associés.

Chaque Category possède un nom qui est sensible à la casse. Pour créer une instance de la classe Category il faut utiliser la méthode statique getInstance() qui attend en paramètre le nom de la Category. Si une Category existe déjà avec le nom fourni, alors la méthode getInstance() renvoie une l'instance existante.

Il est pratique de fournir le nom complet de la classe comme nom de la Category dans laquelle elle est instanciée mais ce n'est pas une obligation. Il est ainsi possible de créer une hiérarchie spécifique différente de celle de l'application, par exemple basée sur des aspects fonctionnels. L'inconvénient d'associer le nom de la classe au nom de la catégorie est qu'il faut instancier un objet Category dans chaque classe : le plus pratique est de déclarer cet objet static.

Exemple :

```
public class Classe1 {
  static Category category = Category.getInstance(Classe1.class.getName());
  ...
}
```

La méthode log(Priority, Object) permet de demander l'émission d'un message associé au niveau de gravité fourni en paramètre. Plusieurs méthodes sont des raccourcis qui évitent d'avoir à préciser le niveau de gravité car celui utilisé sera automatiquement celui associé à la méthode (debug(Object), info(Object), warn(Object), error(Object), fatal(Object)).

Toutes ces méthodes possèdent une surcharge qui attend en paramètre supplémentaire un objet de type Throwable. Ces méthodes ajouteront automatiquement au message la pile d'appels (stacktrace) de l'exception.

La demande est traitée en fonction de la hiérarchie de la Category et de la priorité du message.

Pour éviter d'éventuels traitements inutiles de création du message, il est possible d'utiliser la méthode isEnabledFor(Priority) pour savoir si la catégorie prend en compte la priorité ou non.

Exemple :

```
import org.apache.log4j.*;

public class TestIsEnabledFor {

  static Category cat1 = Category.getInstance(TestIsEnabledFor.class.getName());

  public static void main(String[] args) {
    int i=1;
    int[] occurrence={10,20,30};

    BasicConfigurator.configure();

    cat1.setPriority(Priority.WARN) ;
    cat1.warn("message de test");

    if(cat1.isEnabledFor(Priority.INFO)) {
      System.out.println("traitement du message de priorité INFO");
      cat1.info("La valeur de l'occurrence "+i+" = " + String.valueOf(occurrence[i]));
    }
    if(cat1.isEnabledFor(Priority.WARN)) {
```

```
      System.out.println("traitement du message de priorité WARN");
      cat1.warn("La valeur de l'occurrence "+i+" = " + String.valueOf(occurrence[i]));
    }

  }
}
```

Résultat :

```
0 [main] WARN TestIsEnabledFor  - message de test
traitement du message de priorit_ WARN
50 [main] WARN TestIsEnabledFor  - La valeur de l'occurrence 1 = 20
```

Le nom de la Category permet d'établir une hiérarchie dans les Categorys : ce nom est composé de mots séparés par un caractère point comme pour les packages. D'ailleurs par simplicité et par convention c'est le nom pleinement qualifié de la classe qui est utilisé.

Il existe toujours une catégorie racine créée par Log4J : pour obtenir une instance de cette Category, il faut utiliser la méthode getRoot() de la classe Category car elle ne possède pas de nom.

La méthode getInstance() de la classe Category renvoie toujours la même instance pour un même nom de catégorie. Si cette instance n'existe pas la méthode la créée sinon elle retourne celle existante.

Le message n'est pris en compte que si son niveau de gravité est supérieur ou égal à celui de la catégorie.

Par défaut, une Category hérite du niveau de gravité de sa Category mère selon la hiérarchie des catégories basée sur leurs noms. Ceci est possible car la Category racine à un niveau de gravité par défaut initialisé à DEBUG.

Par exemple, la catégorie com.jmdoudoux.test.log4j hérite des caractéristiques de la catégorie com.jmdoudoux.test.

Il est possible d'associer un niveau de gravité à la Category de façon statique en utilisant la méthode setPriority().

26.2.2.3. La hiérarchie dans les Categorys

Le nom de la catégorie permet de la placer dans une hiérarchie dont la racine est une Category spéciale nommée root qui est créée par défaut sans nom.

La classe Category possède une classe statique getRoot() pour obtenir la Category racine.

La hiérarchie des noms est établie grâce à la notation par point comme pour les packages. D'ailleurs par convention, le nom de la Category correspond généralement au nom pleinement qualifié de la classe qui va utiliser la Category.

Exemple : soit trois catégories
categorie1 nommée "org"
categorie2 nommée "org.moi"
categorie3 nommée "org.moi.projet"

Categorie3 est fille de categorie2, elle même fille de categorie1.

Cette relation hiérarchique est importante car la configuration établie pour une Category est automatiquement propagée par défaut aux Categorys enfants.

L'ordre de la création des Categorys de la hiérarchie ne doit pas obligatoirement respecter l'ordre de la hiérarchie. Celle ci est constituée au fur et à mesure de la création des Categorys.

26.2.3. La gestion des logs à partir de la version 1.2

Les classes Category et Priority sont deprecated est remplacées respectivement par les classes Logger et Level.

26.2.3.1. Les niveaux de gravité : la classe Level

A partir de la version 1.2 de Log4j, la classe Priority ne doit plus être utilisée : il est préférable d'utiliser sa classe fille Level.

Attention la classe Priority n'est pas marquée deprecated car la classe Level en hérite.

La classe org.apache.log4j.Level encapsule donc un niveau de gravité.

Log4j définit plusieurs niveaux de gravité en standard possédant un ordre hiérarchique :

- TRACE : correspond à des messages de traces d'exécution (depuis la version 1.2.12)
- DEBUG : correspond à des messages de débogage
- INFO : correspond à des messages d'information
- WARN : correspond à des messages d'avertissement
- ERROR : correspond à des messages d'erreur
- FATAL : correspond à des messages liés à un arrêt imprévu de l'application

Deux autres niveaux particuliers sont définis et utilisés dans la configuration :

- OFF : aucun niveau de gravité n'est pris en compte
- ALL : tous les niveaux de gravité sont pris en compte

Il est possible de définir ces propres niveaux de gravité en créant une classe qui hérite de la classe Level.

Le choix du niveau de gravité associé à un message est très important. Voici quelques exemples d'utilisation selon chaque niveau de gravité :

Niveau de gravité	Exemple d'utilisation
TRACE	Entrée et sortie de méthodes
DEBUG	Affichage de valeur de données
INFO	Chargement d'un fichier de configuration, début et fin d'exécution d'un traitement long
WARN	Erreur de login, données invalides
ERROR	Toutes les exceptions capturées qui n'empêchent pas l'application de fonctionner
FATAL	Indisponibilité d'une base de données, toutes les exceptions qui empêchent l'application de fonctionner

26.2.3.2. La classe Logger

A partir de la version 1.2 de Log4j, la classe Category ne doit plus être utilisée : il est préférable d'utiliser sa classe fille Logger.

Attention la classe Category n'est pas marquée deprecated car la classe Logger en hérite.

La classe org.apache.log4j.Logger permet donc comme la classe Category de demander l'envoi d'un message dans le système de logs. Un logger compare son niveau de gravité avec celui du message : si ce dernier est supérieur ou égal à celui du logger alors le message est traité.

Un logger est associé à un ou plusieurs appenders : si le message est à traité, celui est envoyé par le logger à ses appenders.

La classe Logger héritant de la classe Category, elle possède toutes ces méthodes notamment celles permettant l'émission d'un message. L'émission de message se fait donc en utilisant la méthode log() ou une des méthodes utilisant implicitement un niveau de gravité (debug(), info(), warn(), error(), fatal()).

Exemple : les deux lignes de code sont équivalentes

Exemple :

```
logger.log(Level.INFO, "mon message");
logger.info("mon message");
```

Pour obtenir une instance de la classe Logger, il faut utiliser sa méthode statique getLogger(). Cette méthode attend en paramètre le nom du logger.

Exemple :

```
package com.jmdoudoux.test.log4j;

import org.apache.log4j.Logger;

public class MaClasse {
  private static final Logger logger = Logger.getLogger("com.jmdoudoux.test.log4j.MaClasse");
}
```

Comme généralement ce nom correspond au nom pleinement qualifié de la classe, une version surchargée de la méthode getLogger() attend en paramètre un objet de type Class pour en extraire le nom.

Exemple :

```
package com.jmdoudoux.test.log4j;

import org.apache.log4j.Logger;

public class MaClasse {
  private static final Logger logger = Logger.getLogger(MaClasse.class);
}
```

La méthode getLogger() permet de s'assurer que pour un même nom cela soit toujours la même instance qui est retournée.

Exemple :

```
package com.jmdoudoux.test.log4j;

import org.apache.log4j.Logger;

public class TestLog4j9 {

  public static void main(String[] args) {
    Logger loggerA = Logger.getLogger("com.jmdoudoux.test.log4j");
    Logger loggerB = Logger.getLogger("com.jmdoudoux.test.log4j");

    System.out.println("loggerA == loggerB : "+(loggerA==loggerB));
  }
}
```

Résultat :

```
loggerA == loggerB : true
```

Le nom de chaque Logger permet de définir une hiérarchie dans les loggers pour permettre de faciliter leur configuration. Cette hiérarchie sur les noms repose sur l'utilisation du caractère point comme pour les packages. Il est dès lors pratique d'utiliser le nom pleinement qualifié de la classe comme nom de logger pour une classe.

Le nom des logger est sensible à la casse.

La hiérarchie commence toujours par un Logger fournit par Log4j : le RootLogger. Pour obtenir une instance de ce logger racine, il faut utiliser la méthode getRootLogger() de la classe Logger.

Le rootLogger a deux caractéristiques distinctives par rapport aux autres loggers :

- Il existe toujours
- Il n'a pas de nom

Lors de la création de l'instance d'un Logger, la hiérarchie est parcourue pour déterminer le Logger le plus proche de la hiérarchie qui est à défaut le rootLogger pour obtenir ces caractéristiques et les reporter sur le nouveau logger.

L'ordre de création des loggers n'a pas d'importance : il n'est pas obligatoire de créer les 1oggers dans leur ordre hiérarchique

Chaque Logger et chaque message possèdent un niveau de gravité. Le Logger compare son niveau de gravité avec celui du message : si le niveau de gravité du message est égal ou supérieur au niveau de gravité du Logger, alors le message est traité par le framework sinon il est ignoré.

Exemple : le message ne sera jamais pris en compte

Exemple :

```
Logger logger = Logger.getLogger("com.jmdoudoux.test.log4j");
logger.setLevel(Level.INFO);
logger.debug("mon message");
```

Chaque logger est associé à un niveau de gravité soit directement soit indirectement par héritage du niveau de gravité de son père dans la hiérarchie. Si le logger ne possède par de niveau de gravité explicite alors c'est celui de son ancêtre le plus proche dans la hiérarchie des loggers.

Comme le logger racine à un niveau de gravité par défaut, cela implique qu'un logger à toujours un niveau de gravité qui lui est associé.

Si aucun logger ne possède de niveau de gravité explicite dans la hiérarchie alors le niveau du logger racine (rootLogger) qui est utilisé. Le rootLogger est toujours définit avec un niveau de gravité qui par défaut est debug.

Il est possible de configurer un logger par programmation.

Il est possible d'associer de façon statique un niveau de gravité au logger en utilisant la méthode setLevel(). Il est cependant préférable d'utiliser la configuration dynamique en utilisant un fichier de configuration qui permet de modifier les paramètres sans modifier le code source.

26.2.3.3. La migration de Log4j antérieure à 1.2 vers 1.2

La migration de l'utilisation des classes Category vers Logger et Priority vers Level peut généralement être fait grâce à un rechercher/remplacer dans le code source :

Rechercher	Remplacer par
Category.getInstance	Logger.getLogger
Category.getRoot	Logger.getRootLogger
Category	Logger

Priority	Level

26.2.4. Les Appenders

La cible de destination de messages est encapsulée dans un objet de type Appender.

L'interface org.apache.log4j.Appender désigne un flux qui représente la log et se charge de l'envoi de messages formatés dans le flux. Le formatage proprement dit est réalisé par un objet de type Layout. Ce layout peut être fourni dans le constructeur adapté ou par la méthode setLayout().

Une Category ou un Logger peut avoir plusieurs appenders. Si la Category ou le Logger décide de traiter la demande d'un message, le message est envoyé à chacun des appenders. Pour ajouter manuellement un appender à une Category, il suffit d'utiliser la méthode addAppender() qui attend en paramètre un objet de type Appender.

L'interface Appender est directement implémentée par la classe abstraite AppenderSkeleton.

Cette classe est la classe mère de toutes les classes fournies avec Log4j pour représenter un type de log. Log4J propose plusieurs appenders en standard :

- AsyncAppender : messages envoyés vers différents appenders de façon périodique et asynchrone
- ExternalyRolledFileAppender : messages envoyés sur un fichier à rotation à la réception d'un message dédié sur une socket et envoi d'un accusé de traitement
- JDBCAppender : messages envoyés dans une base de données (attention à son utilisation)
- JMSAppender : messages envoyés vers une destination utilisant JMS
- LF5Appender : messages envoyés sur une application Swing dédiée
- NTEventLogAppender : messages envoyés dans la log des événements système sur Windows à partir NT
- NullAppender : messages ignorés
- SMTPAppender : messages envoyés par mail
- SocketAppender : messages envoyés dans une socket
- SocketHubAppender : messages envoyés dans plusieurs sockets
- SyslogAppender : messages envoyés dans le démon syslog d'un système Unix
- TelnetAppender : messages envoyés dans une socket en lecture seule facilement consultable avec l'outil telnet
- WriterAppender : cette classe possède deux classes filles : ConsoleAppender et FileAppender.
- ConsoleAppender : messages envoyés sur la console
- FileAppender : messages envoyés dans un fichier. La classe FileAppender possède deux classes filles : DailyRollingAppender et RollingFileAppender
- DailyRollingFileAppender : messages envoyés dans un fichier à rotation périodique (pas obligatoirement journalière)
- RollingFileAppender : messages envoyés dans un fichier à rotation selon sa taille

Pour créer un appender par programmation, il suffit d'instancier un objet d'une de ces classes.

Chaque appender possède des paramètres de configuration dédiés.

Comme un Logger peut avoir plusieurs appenders, un même message peut être envoyé vers plusieurs appenders selon la configuration. La méthode addAppender() de la classe Logger permet d'ajouter manuellement un appender au logger.

Comme pour les niveaux de gravité, les appenders d'une catégorie ou d'un logger sont hérités implicitement par défaut de la hiérarchie des loggers.

Il est possible d'inhiber cet héritage pour une partie de la hiérarchie en utilisant la méthode setAdditivity() avec le paramètre false sur l'instance du logger concerné. Ce logger et sa hiérarchie descendante n'hériteront pas des caractéristiques de leur parent.

Exemple :

```
package com.jmdoudoux.test.log4j;
```

```
import java.io.IOException;

import org.apache.log4j.ConsoleAppender;
import org.apache.log4j.FileAppender;
import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.log4j.SimpleLayout;
import org.apache.log4j.xml.XMLLayout;

public class TestLog4j10 {

  public static void main(
      String[] args) {
    Logger logRoot = Logger.getRootLogger();
    ConsoleAppender ca = new ConsoleAppender();
    ca.setName("console");
    ca.setLayout(new SimpleLayout());
    ca.activateOptions();
    logRoot.addAppender(ca);
    logRoot.setLevel(Level.DEBUG);

    logRoot.debug("message 1");

    Logger log = Logger.getLogger(TestLog4j10.class);

    log.setAdditivity(false);
    try {
      FileAppender fa = new FileAppender(new XMLLayout(), "c:/log.txt");
      fa.setName("FichierLog");
      log.addAppender(fa);
    } catch (IOException e) {
      e.printStackTrace();
    }

    log.debug("message 2");

    Logger logTest = Logger.getLogger("com.jmdoudoux.test.log4j");
    logTest.debug("message 3");
  }

}
```

Résultat dans la console :

```
DEBUG - message 1
DEBUG - message 3
```

Résultat dans le fichier de log :

```
<log4j:event logger="com.jmdoudoux.test.log4j.TestLog4j10" timestamp="1231923298709"
      level="DEBUG" thread="main">
<log4j:message><![CDATA[message 2]]></log4j:message>
</log4j:event>
```

La plupart des appenders nécessitent un appel à leur méthode activateOptions() lorsqu'ils sont configurés par programmation avant qu'ils ne puissent être utilisés.

Il est possible de définir son propre appender en définissant une classe qui implémente l'interface Appender ou qui hérite de la classe AppenderSkeleton.

26.2.4.1. AsyncAppender

La classe org.apache.log4j.AsyncAppender envoie les messages vers différents appenders de façon périodique et asynchrone. Cet appender utilise son propre thread.

Cet appender n'est configurable que dans un fichier de configuration au format XML.

Un tag fils <appender-ref> permet de préciser un appender vers lequel les messages seront envoyés. L'attribut ref permet de préciser le nom de l'appender concerné.

L'attribut bufferSize permet de préciser le nombre de messages qui seront stockés dans le tampon.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration debug="false"
  xmlns:log4j="http://jakarta.apache.org/log4j/">
  <appender name="console" class="org.apache.log4j.ConsoleAppender">
    <param name="Target" value="System.out" />
    <layout class="org.apache.log4j.SimpleLayout" />
  </appender>
  <appender class="org.apache.log4j.FileAppender" name="file">
    <param name="file" value="c:/monapp.log" />
    <layout class="org.apache.log4j.SimpleLayout" />
  </appender>
  <appender class="org.apache.log4j.AsyncAppender" name="async">
    <param name="bufferSize" value="2" />
    <appender-ref ref="file" />
    <appender-ref ref="console" />
  </appender>
  <root>
    <level value="info" />
    <appender-ref ref="async" />
  </root>
</log4j:configuration>
```

26.2.4.2. JDBCAppender

La classe org.apache.log4j.jdbc.JDBCAppender envoie les messages dans une base de données.

Cet appender possède plusieurs propriétés notamment pour préciser les paramètres de connexion à la base de données.

Nom	Rôle
BufferSize	Nombre de messages stockés dans le tampon avant l'insertion dans la base de données
Driver	Pilote JDBC pour l'accès à la base de données
Url	Url de connexion à la base de données
Password	mot de passe de connexion
User	utilisateur de connexion
Sql	requête SQL pour insérer une occurrence dans la base de données

La propriété Sql permet de définir la requête SQL qui permet l'insertion des informations sur le message dans la base de données. La requête doit utiliser les séquences utilisées par le layout PatternLayout

Exemple :

```
INSERT INTO log(dthr, niveau, message) VALUES('%d', '%p', '%m');".
```

Attention : l'utilisation de cet appender fourni par Log4J n'est pas recommandée. Pour plus d'informations consultez la documentation de l'API.

26.2.4.3. JMSAppender

La classe org.apache log4j.net.JMSAppender envoie les message vers une destination JMS.

26.2.4.4. LF5Appender

La classe org.apache.log4j.lf5.LF5Appender envoie les messages sur une application Swing dédiée.

26.2.4.5. NTEventLogAppender

La classe org.apache.log4j.nt.NTEventLogAppender envoie les messages dans la log des événements système sur Windows à partir de Windows NT

26.2.4.6. NullAppender

La classe org.apache.log4j.varia.NullAppender ignore les messages qui lui sont envoyés.

La seule propriété d'un NullAppender est :

Nom	Rôle
Threshold	Limiter les messages pris en compte par l'appender à ceux dont le niveau de gravité est supérieur ou égal à celui de threshold. Ceci vient en plus du niveau de gravité associé au logger. Héritée de AppenderSkeleton

26.2.4.7. SMTPAppender

La classe org.apache.log4j.net.SMTPAppender envoie les messages par mail.

La classe SMTPAppender possède plusieurs attributs :

Nom	Rôle
BufferSize	Nombre de message inclus dans un mail
SMTPHost	nom de la machine qui héberge le serveur SMTP
From	email de l'émetteur du mail
To	email du ou des destinataires du mail
Subject	sujet du mail
Cc	email du ou des destinataires en copie du mail
Bcc	email du ou des destinataires en copie caché du mail
SMTPPassword	mot de passe
SMTPUsername	utilisateur

Par défaut, seuls les messages avec un niveau de gravité supérieur ou égal à ERROR sont traités par cet appender.

Cet appender requiert les bibliothèques JavaBeans Activation Framework et JavaMail pour fonctionner.

26.2.4.8. SocketAppender

La classe org.apache.log4j.net.SocketAppender envoie les messages dans une socket utilisant TCP/IP.

Les données envoyées sont des objets de type LoggingEvent sérialisés.

La classe SocketAppender possède plusieurs attributs :

Nom	Rôle
LocationInfo	Booléen qui précise si des informations de localisation sont envoyées. Par défaut la valeur est false.
Port	port de la machine hôte à utiliser.
RemoteHost	chaîne de caractères qui précise la machine hôte

La méthode activateOptions() permet de réaliser la connexion.

Exemple :

```
package com.jmdoudoux.test.log4j;

import org.apache.log4j.Logger;
import org.apache.log4j.SimpleLayout;
import org.apache.log4j.net.SocketAppender;

public class TestLog4j18 {
  static Logger logger = Logger.getLogger(TestLog4j18.class);

  public static void main(
      String args[]) {
    SocketAppender appender = null;
    try {
      appender = new SocketAppender();
      appender.setPort(10256);
      appender.setRemoteHost("localhost");
      appender.setLocationInfo(true);
      appender.setLayout(new SimpleLayout());
      appender.activateOptions();
    } catch (Exception e) {
      e.printStackTrace();
    }
    logger.addAppender(appender);

    while (true) {
      System.out.println("envoie log");
      logger.debug("msg de debogage");
      logger.info("msg d'information");
      logger.warn("msg d'avertissement");
      logger.error("msg d'erreur");
      logger.fatal("msg d'erreur fatale");
      try {
        Thread.sleep(5000);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
  }
}
```

La configuration dans le fichier properties est similaire aux autres appenders.

Exemple :

```
log4j.appender.socket=org.apache.log4j.net.SocketAppender
log4j.appender.socket.RemoteHost=localhost
log4j.appender.socket.Port=10256
```

```
log4j.appender.socket.LocationInfo=true
```

La configuration dans le fichier XML est similaire aux autres appenders.

Exemple :

```
...
  <appender name="socket" class="org.apache.log4j.net.SocketAppender">
    <param name="Port" value="10256"/>
    <param name="RemoteHost" value="localhost"/>
    <param name="LocationInfo" value="true"/>
  </appender>
...
```

26.2.4.9. SocketHubAppender

La classe org.apache.log4j.net.SocketHubAppender envoie les messages dans plusieurs sockets.

26.2.4.10. SyslogAppender

La classe org.apache.log4j.net.SyslogAppender envoie les messages dans le démon syslog d'un système Unix

26.2.4.11. TelnetAppender

La classe org.apache.log4j.net.TelnetAppender envoie les messages dans une socket en lecture seule facilement consultable avec l'outil telnet.

26.2.4.12. WriterAppender

La classe org.apache.log4j.WriterAppender possède deux classes filles : ConsoleAppender et FileAppender. La classe FileAppender possède deux classes filles : DailyRollingAppender et RollingFileAppender.

Elle possède plusieurs propriétés dont :

Nom	Rôle	Valeur par défaut
Encoding	Préciser le jeu de caractères à utiliser.	null
ImmediateFlush	Préciser si le tampon doit être vidé à chaque opération (pas de mise dans un tampon).	true

Exemple :

```
package com.jmdoudoux.test.log4j;

import java.io.FileOutputStream;

import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.log4j.WriterAppender;
import org.apache.log4j.xml.XMLLayout;

public class TestLog4j11 {
```

```
  public static void main(
      String[] args) {
    Logger logRoot = Logger.getRootLogger();

    WriterAppender appender = null;
    try {
      appender = new WriterAppender(new XMLLayout(), new FileOutputStream("c:/malog.txt"));

    } catch (Exception e) {
      e.printStackTrace();
    }

    logRoot.addAppender(appender);
    logRoot.setLevel(Level.DEBUG);

    logRoot.debug("mon message");
  }
}
```

Résultat : le contenu du fichier c:\malog.txt

```
<log4j:event logger="root" timestamp="1219683683344" level="DEBUG" thread="main">
<log4j:message><![CDATA[mon message]]></log4j:message>
</log4j:event>
```

26.2.4.13. ConsoleAppender

La classe org.apache.log4j.ConsoleAppender envoie les messages sur la console : soit sur la sortie standard (System.out) par défaut soit vers la sortie d'erreurs (System.err).

Les propriétés d'un ConsoleAppender sont :

Nom	Rôle	Valeur par défaut
Encoding	Préciser le jeu de caractères à utiliser. Héritée de WriterAppender	null
ImmediateFlush	Envoyer les messages immédiatement vers la console (pas de mise dans un tampon). Héritée de WriterAppender	true
Target	System.out ou System.err	System.out
Threshold	Limiter les messages pris en compte par l'appender à ceux dont le niveau de gravité est supérieur ou égal à celui de threshold. Ceci vient en plus du niveau de gravité associé au logger. Héritée de AppenderSkeleton	

Exemple :

```
package com.jmdoudoux.test.log4j;

import org.apache.log4j.ConsoleAppender;
import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.log4j.SimpleLayout;

public class TestLog4j12 {

  public static void main(
      String[] args) {
    Logger logRoot = Logger.getRootLogger();
    ConsoleAppender ca = new ConsoleAppender();
    ca.setName("console");
    ca.setLayout(new SimpleLayout());
    ca.activateOptions();
    logRoot.addAppender(ca);
    logRoot.setLevel(Level.DEBUG);
```

```
    logRoot.info("mon message");
  }
}
```

Résultat :

```
2008-06-15 10:22:02,925 [INFO ] (TestLog4j12.java:main:20) mon message
INFO - mon message
```

26.2.4.14. FileAppender

La classe org.apache.log4j.FileAppender envoie les messages dans un fichier.

Les propriétés d'un FileAppender sont :

Nom	Rôle	Valeur par défaut
ImmediateFlush	Envoyer les messages immédiatement vers la console (pas de mise dans un tampon). Héritée de WriterAppender	true
Threshold	Limiter les messages pris en compte par l'appender à ceux dont le niveau de gravité est supérieur ou égal à celui de threshold. Ceci vient en plus du niveau de gravité associé au logger. Héritée de AppenderSkeleton	
Append	Ajouter le message à la fin du fichier ou remplacer le contenu du fichier	True
Encoding	Jeu de caractères utilisé pour l'encodage	
BufferedIO	Préciser si un tampon doit être utilisé	False
BufferSize	Préciser la taille du tampon s'il est utilisé	
File	Nom du fichier	

Exemple :

```
package com.jmdoudoux.test.log4j;

import org.apache.log4j.FileAppender;
import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.log4j.SimpleLayout;

public class TestLog4j13 {

  public static void main(
      String[] args) {
    Logger logRoot = Logger.getRootLogger();

    FileAppender appender = null;
    try {
      appender = new FileAppender();

      appender.setLayout(new SimpleLayout());
      appender.setFile("c:/app_log.txt");
      appender.activateOptions();
      logRoot.addAppender(appender);
      logRoot.setLevel(Level.DEBUG);

      logRoot.info("mon message");

    } catch (Exception e) {
      e.printStackTrace();
    }
  }
```

```
}
```

26.2.4.15. DailyRollingFileAppender

La classe org.apache.log4j.DailyRollingFileAppender envoie les messages dans un fichier à rotation périodique (qui n'est pas obligatoirement journalière).

Les propriétés d'un DailyRollingFileAppender sont :

Nom	Rôle	Valeur par défaut
ImmediateFlush	Envoyer les messages immédiatement vers le fichier (pas de mise dans un tampon). Héritée de WriterAppender	true
Threshold	Limiter les messages pris en compte par l'appender à ceux dont le niveau de gravité est supérieur ou égal à celui de threshold. Ceci vient en plus du niveau de gravité associé au logger. Héritée de AppenderSkeleton	
Append	Ajouter le message à la fin du fichier ou remplacer le contenu du fichier. Héritée de FileAppender	True
Encoding	Préciser le jeu de caractères utilisé pour l'encodage. Héritée de WriterAppender	
BufferedIO	Préciser si un tampon doit être utilisé. Héritée de FileAppender	False
BufferSize	Préciser la taille du tampon s'il est utilisé. Héritée de FileAppender	
File	Nom du fichier. Héritée de FileAppender	
DatePattern	Définir la périodicité de rotation et le suffixe des noms des fichiers créés à chaque rotation	

La valeur de la propriété DatePattern suit le format utilisé par la classe SimpleDateFormat.

Exemple :

'.'yyyy-MM: rotation chaque mois

'.'yyyy-ww: rotation chaque semaine

'.'yyyy-MM-dd: rotation chaque jour à minuit

'.'yyyy-MM-dd-a: rotation chaque jour à midi et à minuit

'.'yyyy-MM-dd-HH: rotation chaque heure

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration debug="false"
  xmlns:log4j="http://jakarta.apache.org/log4j/">
  <appender name="LoggerFile"
    class="org.apache.log4j.DailyRollingFileAppender">
    <param name="File"
      value="c:/monapp.log" />
    <param name="DatePattern" value="'.'yyyy-MM-dd" />
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern"
        value="%d{yyyy-MM-dd HH:mm:ss.SSS} %-8p [%t]:%C - %m%n" />
    </layout>
  </appender>
```

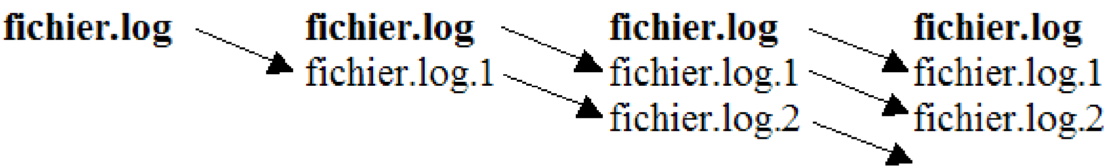
```
  <root>
    <level value="info" />
    <appender-ref ref="LoggerFile" />
  </root>
</log4j:configuration>
```

26.2.4.16. RollingFileAppender

La classe org.apache.log4j.DailyRollingFileAppender envoie les messages dans un fichier à rotation selon sa taille.

Le fichier est créé et rempli avec les différents messages. Une fois que la taille du fichier à atteint celle précisée, le fichier est renommé avec le suffixe .1 et le fichier est recréé. Une fois qu'il est de nouveau rempli, le fichier avec le suffixe .1 est renommé avec .2, le fichier est renommé avec le suffixe .1 et un nouveau fichier est créé.

Si le fichier le plus ancien possède un suffixe supérieur à celui précisé, alors il est supprimé.



Les propriétés d'un RollingFileAppender sont :

Nom	Rôle	Valeur par défaut
ImmediateFlush	Envoyer les messages immédiatement vers le fichier (pas de mise dans un tampon). Héritée de WriterAppender	true
Threshold	Limiter les messages pris en compte par l'appender à ceux dont le niveau de gravité est supérieur ou égal à celui de threshold. Ceci vient en plus du niveau de gravité associé au logger. Héritée de AppenderSkeleton	
Append	Ajouter le message à la fin du fichier ou remplacer le contenu du fichier. Héritée de FileAppender	True
Encoding	Préciser le jeu de caractères utilisé pour l'encodage. Héritée de WriterAppender	
BufferedIO	Préciser si un tampon doit être utilisé. Héritée de FileAppender	False
BufferSize	Préciser la taille du tampon s'il est utilisé. Héritée de FileAppender	
File	Nom du fichier. Héritée de FileAppender	
MaxFileSize	Taille maximale du fichier avant sa rotation. La taille peut être fournie en KB, MB ou GB Exemple : 1024KB	
MaxIndexBackup	Indiquer le nombre de fichier de sauvegarde conservé. Une fois ce nombre dépassé, le dernier fichier de sauvegarde est supprimé	

26.2.4.17. ExternalyRolledFileAppender

La classe org.apache.log4j.ExternalyRollingFileAppender envoie les messages dans un fichier à rotation déclenchée par la réception dans une socket de la chaîne de caractères "RollOver" en respectant la casse .

L'appender envoi en retour un accusé de traitement ou d'erreur via la socket.

La classe ExternalyRolledFileAppender possède plusieurs attributs :

Nom	Rôle	Valeur par défaut
ImmediateFlush	Envoyer les messages immédiatement vers le fichier (pas de mise dans un tampon). Héritée de WriterAppender	true
Threshold	Limiter les messages pris en compte par l'appender à ceux dont le niveau de gravité est supérieur ou égal à celui de threshold. Ceci vient en plus du niveau de gravité associé au logger. Héritée de AppenderSkeleton	
Append	Ajouter le message à la fin du fichier ou remplacer le contenu du fichier. Héritée de FileAppender	True
Encoding	Préciser le jeu de caractères utilisé pour l'encodage. Héritée de WriterAppender	
BufferedIO	Préciser si un tampon doit être utilisé. Héritée de FileAppender	False
BufferSize	Préciser la taille du tampon s'il est utilisé. Héritée de FileAppender	
File	Nom du fichier. Héritée de FileAppender	
MaxFileSize	Taille maximale du fichier avant sa rotation. La taille peut être fournie en KB, MB ou GB Exemple : 1024KB	
MaxIndexBackup	Indiquer le nombre de fichier de sauvegarde conservé. Une fois ce nombre dépassé, le dernier fichier de sauvegarde est supprimé	
Port	Port d'écoute utilisé par la socket	

26.2.5. Les layouts

Ces composants représentés par la classe org.apache.log4j.Layout permettent de définir le format du message avant leur envoie vers leur cible de destination. Un layout est associé à un Appender lors de son instanciation.

Il existe plusieurs layouts définis par log4j :

- HTMLLayout : formate le message en HTML dans un tableau contenant les colonnes (date/heure, niveau de gravité, thead, logger et message)
- PatternLayout : layout le plus puissant puisqu'il permet de préciser le format du message grâce à un motif
- SimpleLayout : layout le plus simple qui ne contient que le niveau de gravité et le message
- XMLLayout : formate le message en XML

Il est possible de créer ses propres layouts en dérivant de la classe Layout.

26.2.5.1. SimpleLayout

La classe org.apache.log4j.SimpleLayout format le message de façon basique en incluant

- le niveau de gravité
- la chaîne de caractère " - "
- et le message

Exemple :

```
package com.jmdoudoux.test.log4j;

import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.log4j.SimpleLayout;
```

```
import org.apache.log4j.FileAppender;

public class TestLog4j8 {
  static Logger logger = Logger.getLogger(TestLog4j8.class);

  public static void main(
      String args[]) {
    SimpleLayout layout = new SimpleLayout();

    FileAppender appender = null;
    try {
      appender = new FileAppender(layout, "c:/monapp.log", false);
    } catch (Exception e) {
      e.printStackTrace();
    }

    logger.addAppender(appender);
    logger.setLevel((Level) Level.DEBUG);

    logger.debug("msg de debogage");
    logger.info("msg d'information");
    logger.warn("msg d'avertissement");
    logger.error("msg d'erreur");
    logger.fatal("msg d'erreur fatale");
  }
}
```

Résultat : le fichier monapp.log

```
DEBUG - msg de debogage
INFO - msg d'information
WARN - msg d'avertissement
ERROR - msg d'erreur
FATAL - msg d'erreur fatale
```

26.2.5.2. HTMLLayout

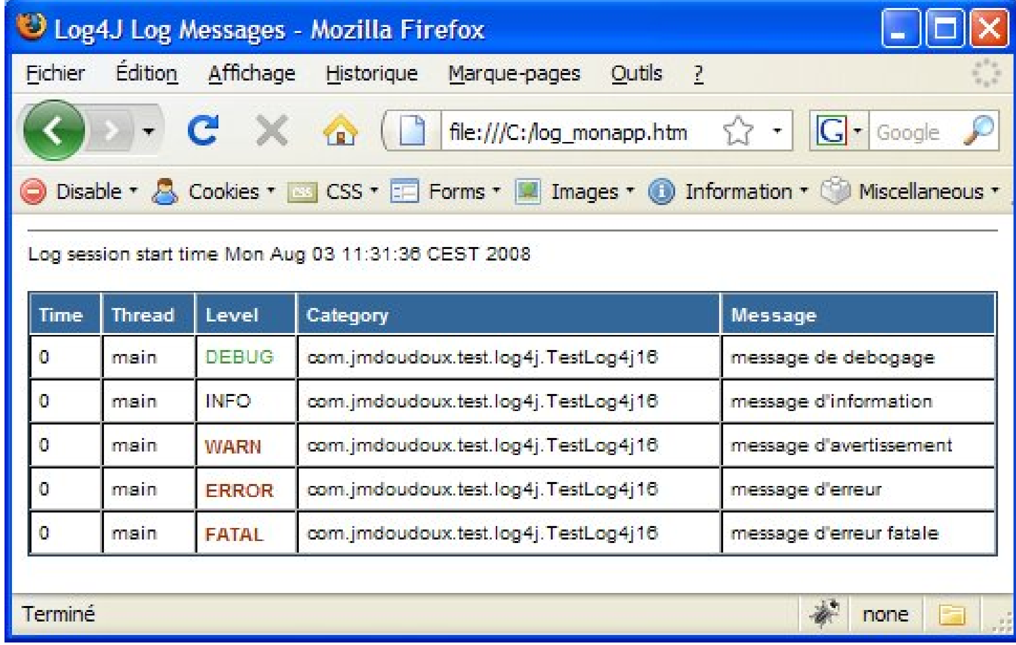
La classe org.apache.log4j.DateLayout format les messages dans un tableau HTML.

Propriété	Rôle	Valeur par défaut
LocationInfo	Inclure des informations sur la classe	False
Title	Précise le titre de la page web	Log4j Log Messages

Exemple :

```
package com.jmdoudoux.test.log4j;

import java.io.*;
import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.log4j.HTMLLayout;
import org.apache.log4j.WriterAppender;

public class TestLog4j16 {
  static Logger logger = Logger.getLogger(TestLog4j16.class);

  public static void main(
      String args[]) {
    HTMLLayout layout = new HTMLLayout();
    WriterAppender appender = null;
    try {
      FileOutputStream output = new FileOutputStream("c:/log_monapp.htm");
      appender = new WriterAppender(layout, output);
    } catch (Exception e) {
      e.printStackTrace();
```

```
    }
    logger.addAppender(appender);
    logger.setLevel((Level) Level.DEBUG);

    logger.debug("msg de debogage");
    logger.info("msg d'information");
    logger.warn("msg d'avertissement");
    logger.error("msg d'erreur");
    logger.fatal("msg d'erreur fatale");
  }
}
```



Log session start time Mon Aug 03 11:31:36 CEST 2008

Time	Thread	Level	Category	Message
0	main	DEBUG	com.jmdoudoux.test.log4j.TestLog4j16	message de debogage
0	main	INFO	com.jmdoudoux.test.log4j.TestLog4j16	message d'information
0	main	WARN	com.jmdoudoux.test.log4j.TestLog4j16	message d'avertissement
0	main	ERROR	com.jmdoudoux.test.log4j.TestLog4j16	message d'erreur
0	main	FATAL	com.jmdoudoux.test.log4j.TestLog4j16	message d'erreur fatale

26.2.5.3. XMLLayout

La classe org.apache.log4j.DateLayout format les messages en XML.

Exemple :

```
<log4j:event logger="com.jmdoudoux.test.log4j.TestLog4j10"
          timestamp="1219683683344" level="DEBUG" thread="main">
<log4j:message><![CDATA[mon message]]></log4j:message>
</log4j:event>
```

26.2.5.4. PatternLayout

Le PatternLayout permet de préciser le format du message grâce à un motif dont certaines séquences seront dynamiquement remplacées par leur valeur correspondante à l'exécution.

Les séquences commencent par un caractère % suivi d'une lettre :

Motif	Rôle
%c	Le nom de la catégorie ou du logger qui a émis le message
%C	Le nom de la classe qui a émis le message : l'utilisation de ce motif est coûteuse en ressources
%d	

	Le timestamp de l'émission du message. Il est possible de fournir un format pour la date/heure en utilisant les motifs de la classe SimpleDateFormat. Exemple : %d{dd MMM yyyy HH:MM:ss } Pour améliorer les performances, il est possible d'utiliser des formateurs de dates en précisant ABSOLUTE, RELATIVE ou ISO8601 Exemple : %d{ABSOLUTE} Sans format précisé, c'est le format défini dans la norme ISO8601 qui est utilisé.
%m	Le message
%n	Un retour chariot dépendant de la plate-forme
%p	Le niveau de gravité du message
%r	Le nombre de millisecondes écoulées entre le lancement de l'application et l'émission du message
%t	Le nom du thread
%x	NDC (Nested Diagnostic Context) du thread. Ceci est particulièrement utile pour les applications de type web.
%%	Le caractère %
%L	Le numéro de ligne dans le code émettant le message : l'utilisation de ce motif est coûteuse en ressources
%F	Le nom du fichier émettant le message : l'utilisation de ce motif est coûteuse en ressources
%M	Le nom de la méthode émettant le message : l'utilisation de ce motif est coûteuse en ressources
%l	Des informations sur l'origine du message dans le code source (C'est un raccourci dépendant de la JVM qui correspond généralement à %C.%M(%F:%L)): l'utilisation de ce motif est coûteux en ressources

Il est possible de préciser le formatage de chaque motif grâce à un alignement et/ou une troncature. Dans le tableau ci dessous, le caractère # représente une des lettres du tableau ci dessus, n représente un nombre de caractères.

Motif	Rôle
%#	Aucun formatage (par défaut)
%n#	Alignement à droite, des blancs sont ajoutés si la taille du motif est inférieure à n caractères
%-n#	Alignement à gauche, des blancs sont ajoutés si la taille du motif est inférieure à n caractères
%.n	Tronque le motif s'il est supérieur à n caractères
%-n.n#	Alignement à gauche, taille du motif obligatoirement de n caractères (troncature ou complément avec des blancs)

Le motif par défaut du PatternLayout est %m%n.

Le motif permet donc une grande souplesse dans le formatage du message.

Exemple :

```
package com.jmdoudoux.test.log4j;

import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.log4j.PatternLayout;
import org.apache.log4j.ConsoleAppender;

public class TestLog4j15 {
  static Logger logger = Logger.getLogger(TestLog4j15.class);
```

```
  public static void main(
      String args[]) {

    StringBuilder motif = new StringBuilder();
    motif.append("Date/heure : %d{yyyy-MM-dd HH:mm:ss.SSS} %n");
    motif.append("Classe emettrice : %C %n");
    motif.append("Localisation : %l %n");
    motif.append("Message: %m %n");
    motif.append("%n");

    PatternLayout layout = new PatternLayout(motif.toString());
    ConsoleAppender appender = new ConsoleAppender(layout);

    logger.addAppender(appender);
    logger.setLevel((Level) Level.DEBUG);

    logger.debug("msg de debogage");
    logger.info("msg d'information");
    logger.warn("msg d'avertissement");
    logger.error("msg d'erreur");
    logger.fatal("msg d'erreur fatale");
  }
}
```

Résultat :

```
Date/heure : 2008-08-03 11:26:13.705
Classe emettrice : com.jmdoudoux.test.log4j.TestLog4j15
Localisation : com.jmdoudoux.test.log4j.TestLog4j15.main(TestLog4j15.java:26)
Message: msg de debogage

Date/heure : 2008-08-03 11:26:13.705
Classe emettrice : com.jmdoudoux.test.log4j.TestLog4j15
Localisation : com.jmdoudoux.test.log4j.TestLog4j15.main(TestLog4j15.java:27)
Message: msg d'information

Date/heure : 2008-08-03 11:26:13.705
Classe emettrice : com.jmdoudoux.test.log4j.TestLog4j15
Localisation : com.jmdoudoux.test.log4j.TestLog4j15.main(TestLog4j15.java:28)
Message: msg d'avertissement

Date/heure : 2008-08-03 11:26:13.705
Classe emettrice : com.jmdoudoux.test.log4j.TestLog4j15
Localisation : com.jmdoudoux.test.log4j.TestLog4j15.main(TestLog4j15.java:29)
Message: msg d'erreur

Date/heure : 2008-08-03 11:26:13.705
Classe emettrice : com.jmdoudoux.test.log4j.TestLog4j15
Localisation : com.jmdoudoux.test.log4j.TestLog4j15.main(TestLog4j15.java:30)
Message: msg d'erreur fatale
```

Voici un exemple de configuration dans un fichier de configuration XML.

Exemple :

```
...
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern"
        value="%d{yyyy-MM-dd HH:mm:ss.SSS} %-8p [%t]:%C - %m%n" />
    </layout>
...
```

Résultat :

```
2008-08-03 09:42:19.342 DEBUG  [main]:com.jmdoudoux.test.log4j.TestLog4j1 - msg de debogage
2008-08-03 09:42:19.342 INFO  [main]:com.jmdoudoux.test.log4j.TestLog4j1 - msg d'information
2008-08-03 09:42:19.342 WARN  [main]:com.jmdoudoux.test.log4j.TestLog4j1 - msg d'avertissement
2008-08-03 09:42:19.342 ERROR [main]:com.jmdoudoux.test.log4j.TestLog4j1 - msg d'erreur
2008-08-03 09:42:19.342 FATAL [main]:com.jmdoudoux.test.log4j.TestLog4j1 - msg d'erreur fatale
```

26.2.6. L'externalisation de la configuration

Log4j peut être entièrement configuré directement dans le code de l'application.

Attention : dans ce cas, la configuration d'un appender nécessite généralement l'appel à la méthode activateOptions() de l'instance de l'Appender pour qu'elle soit prise en compte.

Cependant Log4j est généralement mis en oeuvre avec une externalisation de sa configuration pour ne pas que ces paramètres soient codés en dur. Ceci permet notamment de pouvoir modifier ces paramètres sans avoir à recompiler le code de l'application et donc d'offrir plus de souplesse dans l'utilisation de Log4j.

La configuration de Log4j commence par la définition du ou des appenders qui seront utilisés. Il faut ensuite définir le ou les logger en leur associant au besoin un ou plusieurs appenders. Pour que Log4j fonctionne, il faut à minima associer un appender au logger racine.

Log4j propose deux formats pour externaliser sa configuration :

- fichier properties : les informations sont fournies sous la forme de paire clé=valeur. Le nom de ce fichier est par défaut log4j.properties
- fichier XML : les informations sont fournies dans un document XML. Le nom de ce fichier est par défaut log4j.xml

Le format XML est plus verbeux mais il est mieux structuré. De plus, certaines fonctionnalités ne sont configurables que dans ce format. C'est donc le format dont l'utilisation est recommandée.

Dans les fichiers de configuration, la valeur d'une propriété peut être initialisée avec une variable d'environnement de la JVM en utilisant la syntaxe ${nom_propriété}

26.2.6.1. Les principes généraux

Dans une application, l'initialisation de Log4j n'a besoin d'être réalisée qu'une seule fois de préférence au lancement de l'application.

La configuration suit la même logique que celle des loggers : il est inutile de définir tous les loggers puisque le principe d'héritage permet automatiquement à un logger d'obtenir les caractéristiques de son ascendant le plus proche pour lequel une configuration particulière a été précisée.

Exemple :

Logger	Niveau de gravité	Affectation par
rootLogger	error	assignation (debug par défaut)
com	error	héritage
com.jmdoudoux	error	héritage
com.jmdoudoux.test	info	assignation
com.jmdoudoux.test.log4j	info	héritage
com.jmdoudoux.test.log4j.MaClasse	debug	assignation

Ceci peut permettre de très finement régler le niveau de gravité des différents éléments qui composent une application que ce soit dans les classes de l'application ou d'une bibliothèque tierce.

La configuration au niveau des appenders utilisés suit aussi une logique hiérarchique mais ce n'est pas de l'héritage mais une additivité. Un appender définit dans un logger s'ajoute à ou aux appenders déjà définis dans les loggers de la hiérarchie père.

26.2.6.1.1. Le mécanisme de recherche de la configuration

Log4j propose par défaut un mécanisme de recherche de sa configuration. Log4j recherche un fichier de configuration dans le classpath car il utilise un classLoader pour cette tâche.

Ce mécanisme de recherche peut être désactivé en positionnant à true la propriété système log4j.defaultInitOverride. Ceci doit être utilisé si le chargement de la configuration est fait manuellement dans l'application.

La propriété système log4j.configuration peut être utilisée pour préciser le nom du fichier de configuration.

Par défaut, Log4j recherche dans le classpath un fichier nommé log4j.xml. Si ce fichier n'est pas trouvé, Log4j recherche un fichier nommé log4j.properties.

Log4j utilise un objet de type org.apache.log4j.spi.Configurator pour charger la configuration.

La propriété log4j.configuratorClass permet de préciser explicitement la classe à utiliser pour charger la configuration.

Par défaut, Log4j utilise un objet de type DomConfigurator pour charger un fichier au format XML sinon c'est un objet de type ProperyConfigurator qui est utilisé pour charger le fichier properties.

Vu le mécanisme par défaut proposé par Log4j, le plus simple est donc de nommer son fichier de configuration log4j.xml ou log4j.properties selon le format de configuration utilisé et de mettre le fichier dans le classpath.

26.2.6.2. Le chargement explicite d'une configuration

Si le mode de fonctionnement par défaut ne répond pas aux besoins, il est possible de demander explicitement le chargement d'une configuration.

Pour effectuer ce chargement, l'API fournit plusieurs classes qui implémentent l'interface Configurator. La classe BasicConfigurator est la classe mère des classes PropertyConfigurator (pour la configuration via un fichier de propriétés) et DOMConfigurator (pour la configuration via un fichier XML).

26.2.6.2.1. La classe BasicConfigurator

La classe org.apache.log4j.BasicConfigurator permet de créer une configuration basique.

Avant la version 1.2 de Log4j, la classe BasicConfigurator permet de configurer la catégorie root avec des valeurs par défaut. L'appel à la méthode configure() ajoute à la catégorie root la priorité DEBUG et un ConsoleAppender vers la sortie standard (System.out) associé à un PatternLayout (TTCC_CONVERSION_PATTERN qui est une constante définie dans la classe PatternLayout).

Exemple :

```
import org.apache.log4j.*;

public class TestBasicConfigurator {
  static Category cat = Category.getInstance(TestBasicConfigurator.class.getName()) ;

  public static void main(String[] args) {
    BasicConfigurator.configure();
    cat.info("Mon message");
  }
}
```

Résultat :

```
0 [main] INFO TestBasicConfigurator  - Mon message
```

A partir de la version 1.2 de Log4j, la méthode configure instancie une configuration dont le rootLogger utilise un appender de type ConsoleAppender et un motif PatternLayout.TTCC_CONVERSION_PATTERN pour le PatternLayout utilisé par cet appender. Le niveau de gravité associé est DEBUG par défaut.

Exemple avec Log4j 1.2 :

Exemple :

```
package com.jmdoudoux.test.log4j;

import org.apache.log4j.Logger;
import org.apache.log4j.BasicConfigurator;

public class TestLog4j14 {
  static Logger logger = Logger.getLogger(TestLog4j14.class);

  public static void main(
      String[] args) {

    BasicConfigurator.configure();

    logger.info("debut");
    System.out.println("traitement");
    logger.debug("maValeur");
    logger.info("fin");
  }
}
```

Résultat :

```
0 [main] INFO
com.jmdoudoux.test.log4j.TestLog4j14  -
debut
traitement
0 [main] DEBUG
com.jmdoudoux.test.log4j.TestLog4j14  -
maValeur
0 [main] INFO
com.jmdoudoux.test.log4j.TestLog4j14  -
fin
```

26.2.6.2.2. La classe PropertyConfigurator

La classe org.apache.log4j.PorpertyConfigurator lit un fichier de configuration au format properties et instancie la configuration correspondante.

La classe PropertyConfigurator permet de configurer Log4j à partir d'un fichier de propriétés ce qui évite la recompilation de classes pour modifier la configuration. La méthode configure() qui attend en paramètre un nom de fichier permet de charger la configuration.

Exemple :

```
import org.apache.log4j.*;

public class TestLogging6 {
   static Category cat = Category.getInstance(TestLogging6.class.getName());

   public static void main(String[] args) {
      PropertyConfigurator.configure("logging6.properties");
      cat.info("Mon message");
```

```
   }
}
```

Exemple : le fichier loggin6.properties de configuration de Log4j

```
# Affecte a la catégorie root la priorité DEBUG et un appender nommé CONSOLE_APP
log4j.rootCategory=DEBUG, CONSOLE_APP
# le appender CONSOL_APP est associé à la console
log4j.appender.CONSOLE_APP=org.apache.log4j.ConsoleAppender
# CONSOLE_APP utilise un PatternLayout qui affiche : le nom du thread, la priorité,
# le nom de la catégorie et le message
log4j.appender.CONSOLE_APP.layout=org.apache.log4j.PatternLayout
log4j.appender.CONSOLE_APP.layout.ConversionPattern= [%t] %p %c - %m%n
```

Résultat :

```
C:\>java TestLogging6
[main] INFO TestLogging6 - Mon message
```

26.2.6.2.3. La classe DOMConfigurator

La classe DOMConfigurator permet de configurer Log4j à partir d'un fichier XML ce qui évite aussi la recompilation de classes pour modifier la configuration. La méthode configure() qui attend un nom de fichier permet de charger la configuration. Cette méthode nécessite un parser XML de type DOM compatible avec l'API JAXP.

Le fichier de configuration au format XML doit respecter la dtd log4j.dtd fourni dans la bibliothèque Log4j.

Exemple :

```
import org.apache.log4j.*;
import org.apache.log4j.xml.*;

public class TestLogging7 {
   static Category cat = Category.getInstance(TestLogging7.class.getName());

   public static void main(String[] args) {
      try {
         DOMConfigurator.configure("logging7.xml");
      } catch (Exception e) {
         e.printStackTrace();
      }
      cat.info("Mon message");
   }
}
```

Exemple : le fichier loggin7.xml de configuration de log4j

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">
  <appender name="CONSOLE_APP" class="org.apache.log4j.ConsoleAppender">
     <layout class="org.apache.log4j.PatternLayout">
     <param name="ConversionPattern" value="[%t] %p %c - %m%n"/>
         </layout>
  </appender>
  <root>
    <priority value ="DEBUG" />
    <appender-ref ref="CONSOLE_APP" />
  </root>
</log4j:configuration>
```

Résultat :

```
C:\j2sdk1.4.0-rc\bin>java TestLogging7
[main] INFO TestLogging7 - Mon message
```

26.2.6.3. Les formats des fichiers de configuration

Le fichier de configuration permet :

- de définir les caractéristiques des différents loggers (ou categories)
- de définir les appenders et leur associer un layout
- d'associer un ou plusieurs appenders à un ou aux loggers (ou categories)

Il est préférable d'utiliser un fichier de configuration plutôt que de configurer les entités de Log4j dans le code source car cette dernière solution implique une modification du code et une recompilation pour être pris en compte.

Le fichier de configuration permet basiquement de définir le niveau de gravité minimum à traiter, les flux de sorties (Appender) et le format des messages (Layout).

Deux formats de fichier de configuration sont proposés par Log4j :

- fichier properties : fichier texte dans lequel la configuration est fournie sous la forme de paires clé=valeur
- fichier xml : la configuration est fournie dans un document xml

L'ordre de déclaration des loggers dans le fichier de configuration n'est pas imposé mais il est préférable de conserver un ordre hiérarchique pour en faciliter la lecture et la compréhension.

Généralement le fichier de configuration est lu et utilisé au lancement de l'application.

Chaque appender possède ses propres propriétés de configuration.

Celles-ci sont définies de façons différentes selon le format du fichier de configuration :

- Dans un fichier properties : log4j.appender.nom_appender.nom_propriété=valeur
- Dans un fichier xml : en utilisant le tag fils <param> du tag <appender>

26.2.6.4. La configuration par fichier properties

La configuration utilisant un fichier properties est historiquement la plus ancienne : de nombreuses applications utilisant Log4j la mettent encore en oeuvre.

Comme pour tous fichiers properties, les lignes qui commencent par un caractère dièse sont des lignes de commentaires et sont donc ignorées.

Plusieurs options de configuration générale peuvent être définies dans le fichier de configuration :

Options	Description
log4j.debug	Booléen qui précise si Log4j doit fournir des informations de debogage sur ses activités de chargement du fichier de configuration et de configuration. La valeur par défaut est false.
log4j.disable	Précise le niveau de gravité minimum des messages pour être traités par tous les loggers/categorys. Remarque : l'option log4j.disableOverrider doit obligatoirement être positionnée à false.
log4j.disableOverride	Doit être positionnée à true pour utiliser l'option log4j.disable. La valeur par défaut est false.

La clé log4j.threshold permet de préciser un niveau minimum de gravité pour tous les loggers ou categorys définis indépendamment du niveau spécifié pour chacun d'eux.

Remarque : l'ordre de déclaration des clés dans le fichier n'a pas d'importance pour la bonne mise en oeuvre de Log4j mais il est cependant recommandé d'utiliser un ordre logique pour faciliter la compréhension du paramétrage.

Une category est définie en utilisant une clé de la forme

log4j.category.nom_category

Un logger est défini en utilisant une clé de la forme

log4j.logger.nom_logger

La category racine est configurée en utilisant une clé de la forme

log4j.categoryLogger

Le logger racine est configuré en utilisant une clé de la forme

log4j.rootLogger

La valeur de ces clés est de la forme niveau_gravité, nom_appender1, nom_appender2, ...

Exemple :

```
# le niveau de gravité debug est associe à la catégorie racine avec deux
# appenders nommés A1 et A2
log4j.rootCategory=DEBUG, A1, A2
```

Exemple :

```
# le niveau de gravité debug est associe au logger racine avec deux
# appenders nommés A1 et A2
log4j.rootLogger=DEBUG, A1, A2
```

Remarque : chaque appender doit avoir un nom unique

Le niveau de gravité est optionnel mais dans le cas ou il n'est pas fourni, il est impératif de laisser la virgule entre le signe = et le nom du premier appender.

Exemple :

```
# le niveau de gravité debug (par défaut) est associe au logger racine
# avec un appender nommé A1
   log4j.rootLogger=, A1
```

Un appender est défini en utilisant une clé de la forme

log4j.appender.nom_appender

La valeur de cette clé est le nom pleinement qualifié de la classe qui encapsule l'appender

Exemple :

```
      # l'appender nommé A1 est de type ConsoleAppender
      log4j.appender.A1=org.apache.log4j.ConsoleAppender
      # l'appender nommé A2 est de type RollingFileAppender
      log4j.appender.A2=org.apache.log4j.RollingFileAppender
```

Le layout d'un appender est précisé en utilisant une clé de la forme :

log4j.appender.nom_appender.layout

La valeur de cette clé est le nom pleinement qualifié de la classe qui encapsule le layout.

Exemple :

```
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
```

Une propriété d'un layout est précisée en utilisant une clé de la forme :

log4j.appender.nom_appender.layout.nom_propriété

La valeur sera fournie à la propriété correspondante par introspection.

Exemple :

```
log4j.appender.A1.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} [%-5p] %c- %m%n
log4j.appender.A2.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} [%-5p] %c- %m%n
log4j.appender.A2.File=monapp.log
log4j.appender.A2.MaxFileSize=1024KB
log4j.appender.A2.MaxBackupIndex=2
```

Pour modifier le niveau de gravité pris en compte par un logger il faut utiliser une clé de la forme log4j.logger.nom_logger. La valeur de cette clé doit être le niveau de gravité minimum qui sera traité par le logger.

Exemple :

```
log4j.logger.com.jmdoudoux.test=INFO
```

Il est possible de supprimer l'additivité des appenders d'un logger en utilisant une clé de la forme log4j.additivity.nom_logger. La valeur est un booléen qui précise l'additivité des appenders (la valeur par défaut est true, il faut mettre false pour la supprimer).

Exemple :

```
log4j.additivity.com.jmdoudoux.test=false
```

Il est possible de fournir comme valeur d'une clé la valeur d'une propriété système définie dans la JVM. Pour obtenir la valeur d'une de ces propriétés, il suffit d'utiliser la syntaxe ${nom_de_la_propriete}

26.2.6.5. La configuration via un fichier XML

La structure des données contenues dans le fichier XML est organisée en plusieurs parties définies dans la DTD log4j.dtd et comprend :

- la définition et la configuration des appenders
- la définition et la configuration des loggers
- la configuration du logger racine

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration debug="false"
  xmlns:log4j="http://jakarta.apache.org/log4j/">
```

```
  <appender name="console" class="org.apache.log4j.ConsoleAppender">
    <param name="Target" value="System.out" />
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern"
        value="%d{yyyy-MM-dd HH:mm:ss.SSS} %-8p [%t]:%C - %m%n" />
    </layout>
  </appender>
  <root>
    <appender-ref ref="console" />
  </root>
</log4j:configuration>
```

Si le fichier n'est pas valide alors une exception est levée durant sa lecture et son traitement.

Exemple :

```
log4j:WARN Fatal parsing error 12 and column 5
log4j:WARN The element type "param" must be terminated by the matching end-tag "</param>".
log4j:ERROR Could not parse url
[file:/C:/Documents%20and%20Settings/jmd/workspace/TestLog4j/bin/log4j.xml].
org.xml.sax.SAXParseException: The element type "param" must be terminated
by the matching end-tag "</param>".
```

Les différents éléments qui composent le fichier de configuration sont détaillés dans les sections suivantes.

26.2.6.5.1. Le format du fichier de configuration XML

Le fichier de configuration commence par un prologue et une déclaration de la dtd.

La structure du document xml qui va contenir la configuration de Log4j est définie dans la dtd log4j.dtd.

Cette dtd est fournie dans la bibliothèque log4j.jar dans le package org.apache.log4j.xml

Exemple :

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
```

La dtd défini l'élément racine comme suit :

Exemple :

```
<!ELEMENT log4j:configuration (renderer*, appender*,plugin*, (category|logger)*,root?,
                              (categoryFactory|loggerFactory)?)>
```

L'élément racine est le tag <configuration> associé à l'espace de nommage log4j.

Exemple :

```
<log4j:configuration xmlns:log4j='http://jakarta.apache.org/log4j/'>
</log4j:configuration>
```

Le tag <configuration> peut avoir

- 0 ou plusieurs tags fils <renderer>
- 0 ou plusieurs tags fils <appender>
- 0 ou plusieurs tags fils <plugin>

* 0 ou plusieurs tags fils <category> ou <logger>
* 0 ou 1 tag fils <root>
* 0 ou 1 tag fils <categoryFactory> ou <loggerFactory>

La définition des éléments doit impérativement respecter cet ordre défini dans la DTD.

La dtd définit trois attributs pour le tag <configuration>.

Exemple :

```
<!ATTLIST log4j:configuration
  xmlns:log4j              CDATA #FIXED "http://jakarta.apache.org/log4j/"
  threshold                (all|trace|debug|info|warn|error|fatal|off|null) "null"
  debug                    (true|false|null)  "null"
  reset                    (true|false) "false">
```

Le tag racine est le tag <configuration> qui possède trois attributs :

* threshold : précise le niveau de gravité minimum pour d'un message soit pris en compte par un logger indépendamment du niveau de gravité associé à ce logger
* debug : la valeur true permet de demander à Log4j de fournir des informations de débogage sur son exécution. La valeur par défaut "null" permet de demander l'utilisation de la valeur interne de Log4j
* reset :

L'attribut debug est particulièrement utile pour comprendre l'utilisation du fichier de configuration et résoudre d'éventuel problème dans son contenu.

26.2.6.5.2. La configuration d'un appender

La configuration d'un appender se fait en utilisant un tag <appender>.

La dtd défini l'élément appender comme suit :

Exemple :

```
<!ELEMENT appender (errorHandler?, param*,
      rollingPolicy?, triggeringPolicy?, connectionSource?,
      layout?, filter*, appender-ref*)>
<!ATTLIST appender
  name          CDATA   #REQUIRED
  class         CDATA   #REQUIRED
>
```

Le tag <appender> possède deux attributs obligatoires :

* name : nom unique dans la configuration de l'appender permettant d'y faire référence
* class : nom pleinement qualifié de la classe qui encapsule l'appender

Le tag fils facultatif <param> permet de fournir un paramètre à l'appender. Chaque appender possède ces propres paramètres. Le tag <param> permet de fournir des valeurs aux propriétés de l'appender dont le nom correspond à l'attribut name et la valeur à l'attribut value.

Exemple :

```
<appender name="console" class="org.apache.log4j.ConsoleAppender">
  <param name="Target" value="System.out"/>
</appender>
```

Le tag facultatif layout permet de préciser le layout associé à l'appender. Le tag <layout> possède l'attribut obligatoire class qui précise le nom pleinement qualifié de la classe qui encapsule le layout.

Exemple :

```
<appender name="console" class="org.apache.log4j.ConsoleAppender">
  <layout class="org.apache.log4j.SimpleLayout"/>
</appender>
```

Des paramètres peuvent aussi être fournis au layout en utilisant un ou plusieurs tags fils <param>. Chaque layout possède ces propres propriétés.

Exemple :

```
<appender name="console" class="org.apache.log4j.ConsoleAppender">
  <layout class="org.apache.log4j.PatternLayout">
    <param name="ConversionPattern" value="=%d{yyyy-MM-dd HH:mm:ss} [%-5p] %c- %m%n" />
  </layout>
</appender>
```

26.2.6.5.3. La configuration des Loggers

La configuration d'un logger se fait en utilisant un tag <logger>.

La dtd défini l'élément logger comme suit :

Exemple :

```
<!ELEMENT logger (level?,appender-ref*)>
<!ATTLIST logger
  name         CDATA   #REQUIRED
  additivity   (true|false) "true"
>
```

Le tag <logger> possède un attribut obligatoire :

- name : précise le nom du logger, généralement correspondant à un nom de package ou de classe pleinement qualifié

Le tag <logger> possède un attribut facultatif :

- additivity : précise si l'additivité des appenders doit être poursuivi ou non. La valeur par défaut est true.

Le tag logger peut avoir deux types de tag fils : <level> et <appender-ref>

Le tag facultatif level permet de préciser le niveau de gravité associé au logger. L'attribut value permet de préciser ce niveau de gravité.

Exemple :

```
<logger name="com.jmdoudoux.test.monapp">
  <level value="info"/>
</logger>
```

Le tag <appender-ref> permet d'associer un nouvel appender au logger en plus de ceux associés par additivité des loggers de hiérarchie supérieure. L'attribut ref permet de préciser le nom de l'appender concerné. La valeur de cet attribut doit correspondre à une valeur d'un attribut name d'un appender défini dans la configuration.

Un tag <logger> peut avoir aucun, un ou plusieurs tags <appender-ref> puisqu'un logger peut avoir plusieurs appenders.

Exemple :

```
<logger name="com.jmdoudoux.test.monapp">
  <appender-ref ref="console" />
  <appender-ref ref="journal" />
</logger>
```

26.2.6.5.4. La configuration du logger racine

La configuration du logger racine se fait en utilisant un tag <root>.

La dtd défini l'élément logger comme suit :

Exemple :

```
<!ELEMENT root (param*, (priority|level)?, appender-ref*)>
```

Sa configuration est similaire à celle des loggers sauf que le tag <root> ne possède pas d'attribut.

Exemple :

```
<root>
  <priority value ="info" />
  <appender-ref ref="console"/>
</root>
```

26.2.6.5.5. Les seuils et les filtres pour les appenders

La propriété threshold permet de définir un seuil minimum de niveau de gravité des messages traités par l'appender.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration debug="false"
  xmlns:log4j="http://jakarta.apache.org/log4j/">
  <appender name="console" class="org.apache.log4j.ConsoleAppender">
    <param name="threshold" value="ERROR" />
    <param name="Target" value="System.out" />
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern"
        value="%d{yyyy-MM-dd HH:mm:ss.SSS} %-8p [%t]:%C - %m%n" />
    </layout>
  </appender>
  <root>
    <appender-ref ref="console" />
  </root>
</log4j:configuration>
```

Résultat :

```
2008-08-03 10:03:11.895 ERROR [main]:com.jmdoudoux.test.log4j.TestLog4j1 - msg d'erreur
2008-08-03 10:03:11.910 FATAL [main]:com.jmdoudoux.test.log4j.TestLog4j1 - msg d'erreur fatale
```

Log4j propose un autre mécanisme plus puissant pour filtrer les messages traités par un appender : les filtres.

Log4j propose plusieurs filtres en standard :

- LevelMatchFilter : filtre uniquement les messages ayant un niveau de gravité particulier
- LevelRangeFilter : filtre pour les messages ayant un niveau de gravité compris entre un niveau minimum et un niveau maximum
- DenyAllFilter : filtre pour refuser tous les messages

Les filtres ne peuvent être utilisés que dans une configuration via un fichier xml.

Le filtre LevelMatchFilter possède plusieurs paramètres :

Paramètres	Rôle
levelToMatch	Précise le niveau de gravité du message pour qu'il soit traité
acceptOnMatch	Booléen qui indique si le message est traité (true) ou rejeté (false) par le filtre

Le filtre LevelMatchFilter traite les messages qui correspondent au filtre mais laisse passer ceux qui ne correspondent pas. Ainsi pour ignorer ces messages, il est nécessaire d'appliquer en plus un filtre de type DenyAllFilter.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration debug="false"
  xmlns:log4j="http://jakarta.apache.org/log4j/">
  <appender name="console" class="org.apache.log4j.ConsoleAppender">
    <param name="Target" value="System.out" />
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern"
        value="%d{yyyy-MM-dd HH:mm:ss.SSS} %-8p [%t]:%C - %m%n" />
    </layout>
    <filter class="org.apache.log4j.varia.LevelMatchFilter">
      <param name="levelToMatch" value="ERROR" />
    </filter>
    <filter class="org.apache.log4j.varia.DenyAllFilter"/>
  </appender>
  <root>
    <appender-ref ref="console" />
  </root>
</log4j:configuration>
```

Résultat :

```
2008-08-03 10:14:49.203 ERROR    [main]:com.jmdoudoux.test.log4j.TestLog4j1 - msg d'erreur
```

Si le filtre DenyAllFilter n'est pas utilisé alors tous les messages sont traités.

Résultat :

```
2008-08-03 10:19:28.612 DEBUG [main]:com.jmdoudoux.test.log4j.TestLog4j1 - msg de debogage
2008-08-03 10:19:28.612 INFO  [main]:com.jmdoudoux.test.log4j.TestLog4j1 - msg d'information
2008-08-03 10:19:28.612 WARN  [main]:com.jmdoudoux.test.log4j.TestLog4j1 - msg d'avertissement
2008-08-03 10:19:28.612 ERROR [main]:com.jmdoudoux.test.log4j.TestLog4j1 - msg d'erreur
2008-08-03 10:19:28.612 FATAL [main]:com.jmdoudoux.test.log4j.TestLog4j1 - msg d'erreur fatale
```

Le filtre LevelRangeFilter possède plusieurs paramètres :

Paramètres	Rôle
levelMin	Précise le niveau de gravité minimal du message pour qu'il soit traité
levelMax	Précise le niveau de gravité maximal du message pour qu'il soit traité

acceptOnMatch	true : le message est traité sans appliquer les autres filtres false : si le niveau de gravité est hors de la plage, alors le message est ignoré sinon le message est soumis aux autres filtres

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration debug="false"
  xmlns:log4j="http://jakarta.apache.org/log4j/">
  <appender name="console" class="org.apache.log4j.ConsoleAppender">
    <param name="Target" value="System.out" />
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern"
        value="%d{yyyy-MM-dd HH:mm:ss.SSS} %-8p [%t]:%C - %m%n" />
    </layout>
    <filter class="org.apache.log4j.varia.LevelRangeFilter">
      <param name="levelMin" value="INFO" />
      <param name="levelMax" value="ERROR" />
    </filter>
  </appender>
  <root>
    <appender-ref ref="console" />
  </root>
</log4j:configuration>
```

Résultat :

```
2008-08-03 10:44:46.636 INFO  [main]:com.jmdoudoux.test.log4j.TestLog4j1 - msg d'information
2008-08-03 10:44:46.636 WARN  [main]:com.jmdoudoux.test.log4j.TestLog4j1 - msg d'avertissement
2008-08-03 10:44:46.636 ERROR [main]:com.jmdoudoux.test.log4j.TestLog4j1 - msg d'erreur
```

Avec les filtres, il est par exemple possible de définir un appender qui traite les messages de debogage et un appender qui traite les autres messages.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd" >
<log4j:configuration>
  <appender name="fichierLog"
    class="org.apache.log4j.RollingFileAppender">
    <param name="maxFileSize" value="1024KB" />
    <param name="maxBackupIndex" value="2" />
    <param name="File" value="c:/monapp.log" />
    <param name="threshold" value="info" />
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern"
        value="%d{yyyy-MM-dd HH:mm:ss.SSS} %-8p [%t]:%C - %m%n" />
    </layout>
  </appender>

  <appender name="fichierDebug"
    class="org.apache.log4j.RollingFileAppender">
    <param name="maxFileSize" value="1024KB" />
    <param name="maxBackupIndex" value="2" />
    <param name="File" value="c:/monapp_debug.log" />
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern"
        value="%d{yyyy-MM-dd HH:mm:ss.SSS} %-8p [%t]:%C - %m%n" />
    </layout>
    <filter class="org.apache.log4j.varia.LevelMatchFilter">
      <param name="levelToMatch" value="DEBUG" />
    </filter>
    <filter class="org.apache.log4j.varia.DenyAllFilter"/>
  </appender>

  <root>
```

```
    <priority value="debug"></priority>
    <appender-ref ref="fichierLog" />
    <appender-ref ref="fichierDebug" />
  </root>
</log4j:configuration>
```

26.2.6.6. log4j.xml versus log4j.properties

La configuration par fichier properties est moins verbeuse que par fichier XML.

Certaines fonctionnalités ne sont pas supportées par la configuration par properties comme l'utilisation des Filters ou des ErrorHandlers. Certains appenders ne sont configurables que par fichier XML.

26.2.6.7. La conversion du format properties en format XML

La conversion d'un fichier de configuration au format properties en un fichier de configuration au format XML doit se faire manuellement.

Voici un premier exemple simple.

Exemple :

```
log4j.rootLogger=info, console
log4j.appender.console=org.apache.log4j.ConsoleAppender
log4j.appender.console.Target=System.out
log4j.appender.console.layout=org.apache.log4j.SimpleLayout
```

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd" >
<log4j:configuration>
  <appender name="console" class="org.apache.log4j.ConsoleAppender">
    <layout class="org.apache.log4j.SimpleLayout" />
  </appender>
  <root>
    <priority value="info" />
    <appender-ref ref="console" />
  </root>
</log4j:configuration>
```

Le second exemple ci dessous utilise deux appenders.

Exemple :

```
log4j.rootLogger=debug, console, fichier

log4j.appender.console=org.apache.log4j.ConsoleAppender
log4j.appender.console.layout=org.apache.log4j.PatternLayout
log4j.appender.console.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss.SSS} %-8p [%t]:%C - %m%n

log4j.appender.fichier=org.apache.log4j.RollingFileAppender
log4j.appender.fichier.File=c:/monapp.log
log4j.appender.fichier.MaxFileSize=1024KB
log4j.appender.fichier.MaxBackupIndex=2
log4j.appender.fichier.layout=org.apache.log4j.PatternLayout
log4j.appender.fichier.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss.SSS} %-8p [%t]:%C - %m%n
```

Exemple :

```
<?xml version="1.0" encoding="UTF-8" ?>
!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">
  <appender name="console" class="org.apache.log4j.ConsoleAppender">
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern"
        value="%d{yyyy-MM-dd HH:mm:ss.SSS} %-8p [%t]:%C - %m%n" />
    </layout>
  </appender>
  <appender name="fichier"
    class="org.apache.log4j.RollingFileAppender">
    <param name="file" value="c:/monapp.log" />
    <param name="MaxFileSize" value="1024KB" />
        <param name="MaxBackupIndex" value="2" /></
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern"
        value="%d{yyyy-MM-dd HH:mm:ss.SSS} %-8p [%t]:%C - %m%n" />
    </layout>
  </appender>
  <root>
    <priority value="debug" />
    <appender-ref ref="console" />
    <appender-ref ref="fichier" />
  </root>
</log4j:configuration></pre>
```

26.2.7. La mise en oeuvre avancée

Cette section présente quelques fonctionnalités avancées de Log4j.

26.2.7.1. La lecture des logs

La consultation des logs peut ne pas être facile si elle doit être réalisée en temps réel ou si le volume de messages est très important.

26.2.7.1.1. La lecture pendant l'exécution du programme

Si l'application écrit régulièrement dans le fichier, un simple bloc note n'est pas suffisant pour lire les messages arrivés après l'ouverture du fichier.

Sous Unix, la commande tail est particulièrement utile car elle permet de visualiser les n dernières lignes d'un fichier alors que celui-ci est en train de grossir.

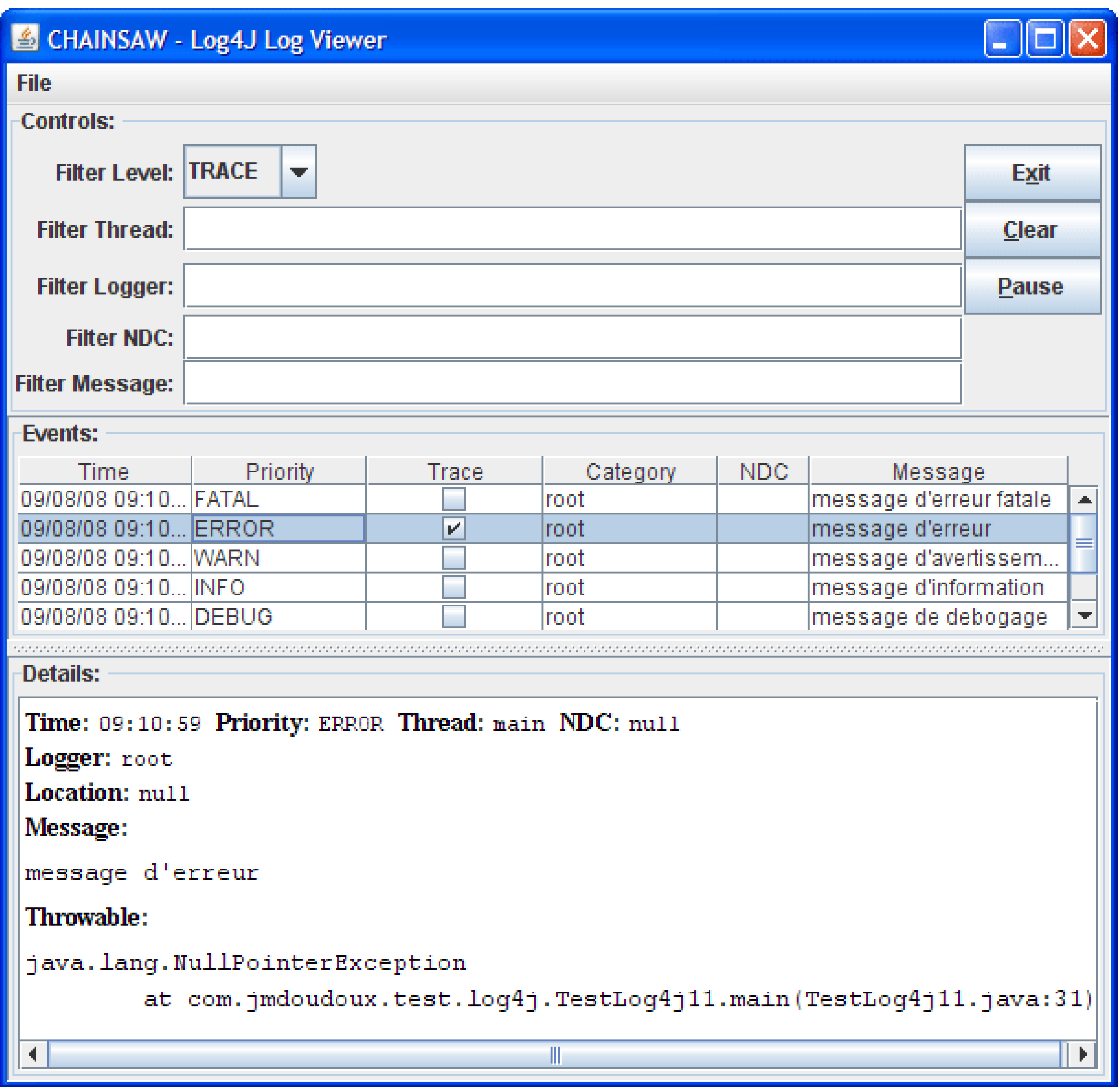
26.2.7.1.2. L'application Chainsaw

Log4j propose en standard une application graphique nommée chainsaw qui permet de visualiser des logs formatées avec un layout XMLLayout ou envoyées par un SocketAppender.

Pour exécuter Chainsaw, il faut exécuter la classe org.apache.log4j.chainsaw.Main

Exemple :

```
C:\>java -cp C:/java/apache-log4j-1.2.15/log4j-1.2.15.jar org.apache.log4j.chainsaw.Main
```



Pour consulter un fichier XML qui contient des logs formatées avec un XML Layout, il faut utiliser l'option "Load File" du menu "File".

La partie "Controls" propose plusieurs filtres : il suffit de saisir les caractères recherchés et le filtre est appliqué au fur et à mesure de la saisie.

ChainSaw est aussi très pratique pour consulter les logs envoyées par un SocketAppender.

Il faut définir une variable d'environnement à la JVM nommée chainsaw.port pour préciser le port à écouter.

Exemple :

```
C:\>java -cp C:/java/apache-log4j-1.2.15/log4j-1.2.15.jar -Dchainsaw.port=10256
org.apache.log4j.chainsaw.Main
```

26.2.7.2. Les variables d'environnement

Log4j utilise plusieurs variables d'environnement de la JVM pour éventuellement modifier certains comportement.

Variable	Rôle
log4j.debug	Fournir des informations de débogage lors de la recherche de la configuration et de son chargement

log4j.configuration	Permet de préciser le nom du fichier properties qui contient la configuration. Ce fichier doit être dans le classpath
log4j.defaultInitOverride	Booléen qui permet de demander d'inhiber la recherche de la configuration. La valeur par défaut est false.

26.2.7.3. L'internationalisation des messages

La classe Category et par héritage la classe Logger proposent deux surcharges de la méthode l7dlog() qui permettent l'émission de messages internationalisés (l7d est le raccourci de localized).

Avant la première utilisation de la méthode l7dlog, il est nécessaire de préciser quel ResourceBundle doit être utilisé en invoquant la méthode setResourceBundle().

Les méthodes l7dlog() attendent en paramètres le niveau de gravité, la clé du message dans le resourceBundle et un objet de type Throwable. La seconde surcharge attend aussi un tableau d'objets qui seront insérés à leur emplacement définis dans la valeur de la clé.

Exemple :

```
package com.jmdoudoux.test.log4j;

import java.util.Locale;
import java.util.ResourceBundle;

import org.apache.log4j.ConsoleAppender;
import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.log4j.SimpleLayout;

public class TestLog4j19 {
  static Logger logger = Logger.getLogger(TestLog4j19.class);

  public static void main(
      String args[]) {
    Logger logRoot = Logger.getRootLogger();
    ConsoleAppender ca = new ConsoleAppender();
    ca.setName("console");
    ca.setLayout(new SimpleLayout());
    ca.activateOptions();
    logRoot.addAppender(ca);
    logRoot.setLevel(Level.DEBUG);

    Locale locale = new Locale("fr", "FR");
    ResourceBundle messages = ResourceBundle.getBundle("MessagesLog", locale);
    logger.setResourceBundle(messages);

    logger.l7dlog(Level.DEBUG, "MESSAGE", null);

    locale = new Locale("en", "EN");
    messages = ResourceBundle.getBundle("MessagesLog", locale);
    logger.setResourceBundle(messages);

    logger.l7dlog(Level.DEBUG, "MESSAGE", null);

  }
}
```

Il faut définir les fichiers properties qui seront utilisés par le ResourceBundle. Ces fichiers doivent être stockés dans le classpath.

Exemple : le fichier MessagesLog.properties

```
MESSAGE=mon message en français
```

Exemple : le fichier MessagesLog_en.properties

```
MESSAGE=my message in english
```

Résultat :

```
DEBUG - mon message en français
DEBUG - my message in english
```

26.2.7.4. L'initialisation de Log4j dans une webapp

Cette section sera développée dans une version future de ce document

26.2.7.5. La modification dynamique de la configuration

Cette section sera développée dans une version future de ce document

26.2.7.6. NDC/MDC

Cette section sera développée dans une version future de ce document

26.2.8. Des best practices

Cette section fournit quelques best practices lors de la mise oeuvre de Log4j.

26.2.8.1. Le choix du niveau de gravité des messages

Le choix du niveau de gravité de chaque message émis est très important.

Voici quelques exemples d'utilisation de chaque niveau de gravité :

- fatal : messages concernant un arrêt imprévu de l'application
- error : messages d'erreurs nécessitant une analyse par exemple les exceptions levées
- warn : message d'avertissement

* info : messages d'information par exemple le démarrage ou l'arrêt de l'application, la connexion à une ressource, ...
* trace : messages pour suivre le flow d'exécution
* debug : messages de débogage par exemple pour obtenir la valeur de variables, ...

Hors de l'environnement de développement, le niveau de gravité minimum des messages doit être info. Le niveau debug n'est à utiliser que dans l'environnement de développement ou à utiliser temporairement pour des besoins spécifiques dans les autres environnements.

26.2.8.2. L'amélioration des performances

Log4j a été développé dans le soucis de réduire au minimum le surcoût de son utilisation.

Cependant le logging a nécessairement un coût et ce coût peut devenir important si certaines précautions ne sont pas prises par le développeur.

Il est nécessaire de limiter le coût d'émission d'un message dont le coût de construction est important surtout si ce dernier sera ignoré par le logger.

Exemple :

```
logger.debug("valeur="+valeur+" , i="+i+" ,next="+next);
```

Dans cet exemple, si le niveau de gravité du logger est supérieur à debug, le coût de l'émission du message contiendra aussi la création du message par concaténation des différentes valeurs.

Pour limiter ce coût, il est préférable de conditionner l'émission de message par un test préalable sur le niveau de gravité pris en charge par le logger lors de l'exécution du traitement.

Les classes Category et Logger proposent des méthodes pour effectuer ces tests.

Exemple :

```
if (logger.isDebugEnabled()) {
  logger.debug("valeur="+valeur+" , i="+i+" ,next="+next);
}
```

Avec ce test, le message n'est construit que s'il est pris en compte par le logger. L'inconvénient de ce test est qu'il est réalisé deux fois : une fois par la méthode isDebugEnabled() et une autre fois par la méthode debug(). Cependant ce surcoût est beaucoup moins important que la création du message.

Les temps de traitement de Log4j sont obligatoirement dépendants de l'utilisation qui en est faite dans l'application notamment :

* plus il a de messages émis plus les traitements sont longs : par exemple, il faut éviter d'envoyer un message dans une boucle
* plus les niveaux de gravité associés à un appender sont bas dans la hiérarchie, plus le nombre de messages à traiter est important
* plus il y a d'appender plus le temps de traitement d'un message est important

Lors de l'utilisation d'un layout de type PatternLayout, l'utilisation de certains motifs sont connus pour être gourmand en temps de traitement. Même si les informations de ces motifs sont particulièrement utiles, il faut tenir compte de leur temps de traitement lors de leur utilisation.

Pour économiser de la mémoire, il est préférable de déclarer les loggers en tant que variables statiques.

Exemple :

```
private static Logger logger = Logger.getLogger(MaClasse.class);
```

26.3. L'API logging

L'usage de fonctionnalités de logging dans les applications est tellement répandu que SUN a décidé de développer sa propre API et de l'intégrer au JDK à partir de la version 1.4.

Cette API a été proposée à la communauté sous la Java Specification Request numéro 047 (JSR-047).

Le but est de proposer un système qui puisse être exploité facilement par toutes les applications.

L'API repose sur cinq classes principales et une interface:

- Logger : cette classe permet d'envoyer des messages sans le système de log
- LogRecord : cette classe encapsule le message
- Handler : cette classe représente la destination qui va recevoir les messages
- Formatter : cette classe permet de formater le message avant son envoi vers la destination
- Filter : cette interface doit être implémentée par les classes dont le but est de déterminer si le message doit être envoyé vers une destination
- Level : cette représente le niveau de gravité du message

Un logger possède un ou plusieurs Handlers qui sont des entités qui vont recevoir les messages. Chaque handler peut avoir un filtre associé en plus du filtre associé au Logger.

Chaque message possède un niveau de sévérité représenté par la classe Level.

26.3.1. La classe LogManager

Cette classe est un singleton qui propose la méthode getLogManager() pour obtenir l'unique référence sur un objet de ce type.

Cette objet permet :

- de maintenir une liste de Logger désigné par un nom unique.
- de maintenir une liste de Handlers globaux
- de modifier le niveau de sévérité pris en compte pour un ou plusieurs Logger dont le début du nom est précisé

Pour réaliser ces actions, la classe LogManager possède plusieurs méthodes dont les principales sont :

Méthode	Rôle
void addGlobalHandler(Handler)	Ajoute un handler à la liste des handler globaux
Logger getLogger(String)	Permet d'ajouter un nouveau Logger dont le nom fourni en paramètre lui sera attribué si il n'existe pas encore sinon la référence sur le Logger qui possède déjà ce nom est renvoyé.
void removeAllGlobalHandlers()	Supprime tous les Handler de la liste de handler globaux
void removeGlobalHandler(Handler)	Supprime le Handler de la liste de Handler globaux
void setLevel(String, Level)	Permet de préciser le niveau de tout les Logger dont le nom commence par la chaîne fournie en paramètre
LogManager getLogManager()	Renvoie l'instance unique de cette classe

Par défaut, la liste des Loggers contient toujours un Logger nommé global qui peut être facilement utilisé.

26.3.2. La classe Logger

La classe Logger est la classe qui se charge d'envoyer les messages dans la log. Un Logger est identifié par un nom qui est habituellement le nom qualifié de la classe dans laquelle le Logger est utilisé. Ce nom permet de gérer une hiérarchie de Logger. Cette gestion est assurée par le LogManager. Cette hiérarchie permet d'appliquer des modifications sur un Logger ainsi qu'à toute sa "descendance".

Il est aussi de possible de créer des Logger anonymes.

La méthode getLogger() de la classe LogManager permet d'instancier un nouvel objet Logger si aucun Logger possédant le nom passé en paramètre a déjà été défini sinon il renvoie l'instance existante.

La classe Logger se charge d'envoyer les messages aux Handlers enregistrés sous la forme d'un objet de type LogRecord. Par défaut, ces Handlers sont ceux enregistrés dans le LogManager. L'envoi des messages est conditionné par la comparaison du niveau de sévérité de message avec celui associé au Logger.

La classe Logger possède de nombreuses méthodes pour générer des messages : plusieurs méthodes sont définies pour chaque niveau de sécurité. Plutôt que d'utiliser la méthode log() en précisant le niveau de sévérité, il est possible d'utiliser la méthode correspondante au niveau de sécurité.

Ces méthodes sont surchargées pour accepter plusieurs paramètres :

- le nom de la classe, le nom de la méthode et le message : les deux premières données sont très utiles pour faciliter le débogage
- le message: dans ce cas, le framework tente de déterminer dynamiquement le nom de la classe et de la méthode

26.3.3. La classe Level

Chaque message est associé à un niveau de sévérité représenté par un objet de type Level. Cette classe définit 7 niveaux de sévérité :

- Level.SEVERE
- Level.WARNING
- Level.INFO
- Level.CONFIG
- Level.FINE
- Level.FINER
- Level.FINEST

26.3.4. La classe LogRecord

Cette section sera développée dans une version future de ce document

26.3.5. La classe Handler

Le framework propose plusieurs classes filles qui représentent différents moyens pour émettre les messages :

- StreamHandler : envoie des messages dans un flux de sortie
- ConsoleHandler : envoie des messages sur la sortie standard d'erreur

* FileHandler : envoie des messages sur un fichier
* SocketHandler : envoie des messages dans une socket réseau
* MemoryHandler : envoie des messages dans un tampon en mémoire

26.3.6. La classe Filter

Cette section sera développée dans une version future de ce document

26.3.7. La classe Formatter

Le framework propose deux implémentations :

* SimpleFormatter : pour formater l'enregistrement sous forme de chaîne de caractères
* XMLFormatter : pour formater l'enregistrement au format XML

XMLFormatter utilise un DTD particulière. Le tag racine est <log>. Chaque enregistrement est inclus dans un tag <record>.

26.3.8. Le fichier de configuration

Un fichier particulier au format Properties permet de préciser des paramètres de configuration pour le système de log tel que le niveau de sévérité géré par un Logger particulier et sa descendance, les paramètres de configuration des Handlers ...

Il est possible de préciser le niveau de sévérité pris en compte par tous les Logger :

.level = niveau

Il est possible de définir les handlers par défaut :

handlers = java.util.logging.FileHandler

Pour préciser d'autre handler, il faut les séparer par des virgules.

Pour préciser le niveau de sévérité d'un Handler, il suffit de le lui préciser :

java.util.logging.FileHandler.level = niveau

Un fichier par défaut est défini avec les autres fichiers de configuration dans le répertoire lib du JRE. Ce fichier ce nomme logging.properties.

Il est possible de préciser un fichier particulier précisant son nom dans la propriété système java.util.logging.config.file

exemple : java -Djava.util.logging.config.file=monLogging.properties

26.3.9. Exemples d'utilisation

Cette section sera développée dans une version future de ce document

26.4. Jakarta Commons Logging (JCL)

Le projet Jakarta Commons propose un sous projet nommé Logging qui encapsule l'usage de plusieurs systèmes de logging et facilite ainsi leur utilisation dans les applications. Ce n'est pas un autre système de log mais il propose un niveau d'abstraction qui permet sans changer le code d'utiliser indifféremment n'importe lequel des systèmes de logging supportés. Son utilisation est d'autant plus pratique qu'il existe plusieurs système de log dont aucun des plus répandus, Log4j et l'API logging du JDK 1.4, ne sont dominants.

Le grand intérêt de cette bibliothèque est donc de rendre l'utilisation d'un système de log dans le code indépendant de l'implémentation de ce système. JCL encapsule l'utilisation de Log4j, l'API logging du JDK 1.4 et LogKit.

De nombreux projets du groupe Jakarta utilise cette bibliothèque tel que Tomcat ou Struts. La version de JCL utilisée dans cette section est le 1.0.3

Le package, contenu dans le fichier commons-logging-1.0.3.zip peut être téléchargé sur le site http://jakarta.apache.org/commons/logging.html. Il doit ensuite être décompressé dans un répertoire du système d'exploitation.

Pour utiliser la bibliothèque, il faut ajouter le fichier dans le classpath.

L'inconvénient d'utiliser cette bibliothèque est qu'elle n'utilise que le dénominateur commun des système de log qu'elle supporte : ainsi certaines caractéristiques d'un système de log particulier ne pourront être utilisées via cette API .

La bibliothèque propose une fabrique qui renvoie, en fonction du paramètre précisé, un objet qui implémente l'interface Log. La méthode statique getLog() de la classe LogFactory permet d'obtenir cet objet : elle attend en paramètre soit un nom sous la forme d'une chaîne de caractères soit un objet de type Class dont le nom sera utilisé. Si un objet de type log possédant ce nom existe déjà alors c'est cette instance qui est renvoyée par la méthode sinon c'est une nouvelle instance qui est retournée. Ce nom représente la catégorie pour le système log utilisé, si celui ci supporte une telle fonctionnalité.

Par défaut, la méthode getLog() utilise les règles suivantes pour déterminer le système de log à utiliser :

- Si la bibliothèque Log4j est incluse dans le classpath de la JVM alors celle ci sera utilisée par défaut par la bibliothèque Commons Logging.
- Si le JDK 1.4 est utilisé et que Log4j n'est pas trouvé alors le système utilisé par défaut est celui fourni en standard avec le JDK (java.util.logging)
- Si aucun de ces systèmes de log n'est trouvé, alors JCL utilise un système de log basic fourni dans la bibliothèque : SimpleLog. La configuration de ce système ce fait dans un fichier nommé simplelog.properties

Il est possible de forcer l'usage d'un système de log particulier en précisant la propriété org.apache.commons.logging.Log à la machine virtuelle.

Pour complètement désactiver le système de log, il suffit de fournir la valeur org.apache.commons.logging.impl.NoOpLog pour la propriété org.apache.commons.logging.Log à la JVM. Attention dans ce cas, plus aucune log ne sera émise.

Il existe plusieurs niveaux de gravité que la bibliothèque tentera de faire correspondre au mieux avec le système de log utilisé.

26.5. D'autres API de logging

Il existe d'autres API de logging dont voici une liste non exhaustive :

Produit	URL
Lumberjack	http://javalogging.sourceforge.net/
Javalog	http://sourceforge.net/projects/javalog/
Jlogger de Javelin Software	http://www.javelinsoft.com/jlogger/

27. La sécurité

La suite de ce chapitre sera développée dans une version future de ce document

Depuis sa conception, la sécurité dans le langage Java a toujours été une grande préoccupation pour Sun.

Avec Java, la sécurité revêt de nombreux aspects :

- les spécifications du langage disposent de fonctionnalité pour renforcer la sécurité du code
- la plate-forme définit un modèle pour gérer les droits d'une application
- l'API JCE permet d'utiliser des technologies de cryptographie
- l'API JSSE permet d'utiliser le réseau au travers des protocoles sécurisés SSL ou TLS
- l'API JAAS propose un service pour gérer l'authentification et les autorisations d'un utilisateur

Ces deux premiers aspects ont été intégrés à java dès sa première version.

Ce chapitre contient plusieurs sections :

- La sécurité dans les spécifications du langage
- Le contrôle des droits d'une application
- JCE (Java Cryptography Extension)
- JSSE (Java Secure Sockets Extension)
- JAAS (Java Authentication and Authorization Service)

27.1. La sécurité dans les spécifications du langage

Les spécifications du langage apportent de nombreuses fonctionnalités pour renforcer la sécurité du code aussi bien lors de la phase de compilation que lors de la phase d'exécution :

- typage fort (toutes les variables doivent posséder un type)
- initialisation des variables d'instances avec des valeurs par défaut
- modificateur d'accès pour gérer l'encapsulation et donc l'accessibilité aux membres d'un objet
- les membres final
- ...

27.1.1. Les contrôles lors de la compilation

27.1.2. Les contrôles lors de l'exécution

La JVM exécute un certain nombre de contrôle au moment de l'exécution :

- vérification des accès en dehors des limites des tableaux
- contrôle de l'utilisation des casts
- vérification par le classloader de l'intégrité des classes utilisées
- ...

27.2. Le contrôle des droits d'une application

Un système de contrôle des droits des applications a été intégré à Java dès sa première version notamment pour permettre de sécuriser l'exécution des applets. Ces applications téléchargées sur le réseau et exécutées sur le poste client doivent impérativement assurer aux personnes qui les utilisent que celles-ci ne risquent pas de réaliser des actions malveillantes sur le système dans lequel elles s'exécutent.

Le modèle de sécurité relatif aux droits des applications développées en Java a évolué au fur et à mesure des différentes versions de Java.

27.2.1. Le modèle de sécurité de Java 1.0

Le modèle proposé par Java 1.0 est très sommaire puisqu'il ne distingue que deux catégories d'applications :

- les applications locales
- les applications téléchargées sur le réseau

Le modèle est basé sur le "tout ou rien". Les applications locales ont tous les droits et les applications téléchargées ont des droits très limités. Les restrictions de ces dernières sont nombreuses :

- impossibilité d'écrire sur le disque local
- impossibilité d'obtenir des informations sur le système local
- impossibilité de se connecter à un autre serveur que celui d'ou l'application a été téléchargée
- ...

La mise en oeuvre de ce modèle est assurée par le "bac à sable" (sand box en anglais) dans lequel s'exécutent les applications téléchargées.

27.2.2. Le modèle de sécurité de Java 1.1

Le modèle proposé par la version 1.0 est très efficace mais beaucoup trop restrictif surtout dans le cadre d'utilisation personnelle tel que des applications pour un intranet par exemple.

Le modèle de la version 1.1 propose la possibilité de signer les applications packagées dans un fichier .jar. Une application ainsi signée possède les mêmes droits qu'une application locale.

27.2.3. Le modèle Java 1.2

Le modèle proposé par la version 1.1 a apporté de début de solution pour attribuer des droits sensibles à certaines applications. Mais ce modèle manque cruellement de souplesse puisqu'il s'appuie toujours sur le modèle "tout au rien".

Le modèle de la version 1.2 apporte enfin une solution très souple mais plus compliquée à mettre en oeuvre.

Les droits accordés à une application sont rassemblés dans un fichier externe au code qui se nomme politique de sécurité. Ces fichiers se situent dans le répertoire lib/security du répertoire ou est installé le JRE. Par convention, ces fichiers ont pour extension .policy.

27.3. JCE (Java Cryptography Extension)

JCE est une API qui propose de standardiser l'utilisation de la cryptographie en restant indépendant des algorithmes utilisés. Elle prend en compte le cryptage/décryptage de données, la génération de clés, et l'utilisation de la technologie MAC (Message Authentication Code) pour garantir l'intégrité d'un message.

JCE a été intégré au JDK 1.4. Auparavant, cette API était disponible en tant qu'extension pour les JDK 1.2 et 1.3.

Pour pouvoir utiliser cette API, il faut obligatoirement utiliser une implémentation développée par un fournisseur (provider). Avec le JDK 1.4, Sun fournit une implémentation de référence nommée SunJCE.

Les classes et interfaces de l'API sont regroupées dans le package javax.crypto

27.3.1. La classe Cipher

Cette classe encapsule le cryptage et le décryptage de données.

La méthode statique getInstance() permet d'obtenir une instance particulière d'un algorithme fourni par un fournisseur. Le nom de l'algorithme est fourni en paramètre de la méthode sous la forme d'une chaîne de caractères.

Avant la première utilisation de l'instance obtenue, il faut initialiser l'objet en utilisant une des nombreuses surcharges de la méthode init().

27.4. JSSE (Java Secure Sockets Extension)

Les classes et interfaces de cette API sont regroupées dans les packages javax.net et javax.net.ssl.

27.5. JAAS (Java Authentication and Authorization Service)

Les classes et interfaces de cette API sont regroupées dans le package javax.security.auth

Cette API a été intégrée au J.D.K. 1.4.

28. JNI (Java Native Interface)

JNI est l'acronyme de Java Native Interface. C'est une technologie qui permet d'utiliser du code natif dans une classe Java notamment C.

L'inconvénient majeur de cette technologie est d'annuler la portabilité du code Java. En contre partie cette technologie peut être très utile dans plusieurs cas :

- pour des raisons de performance
- utiliser des composants éprouvés déjà existants

La mise en oeuvre de JNI nécessite plusieurs étapes :

- la déclaration et l'utilisation de la ou des méthodes natives dans la classe Java
- la compilation de la classe Java
- la génération du fichier d'en-tête avec l'outil javah
- l'écriture du code natif en utilisant entre autre les fichiers d'en-tête fourni par le JDK et celui généré précédemment
- la compilation du code natif sous la forme d'une bibliothèque

Le format de la bibliothèque est donc dépendante du système d'exploitation pour lequel elle est développée : .dll pour les systèmes de type Windows, .so pour les systèmes de type Unix, ...

Ce chapitre contient plusieurs sections :

- La déclaration et l'utilisation d'une méthode native
- La génération du fichier d'en-tête
- L'écriture du code natif en C
- Le passage de paramètres et le renvoi d'une valeur (type primitif)
- Le passage de paramètres et le renvoi d'une valeur (type objet)

28.1. La déclaration et l'utilisation d'une méthode native

La déclaration dans le code source Java est très facile puisqu'il suffit de déclarer la signature de la méthode avec le modificateur native. Le modificateur permet au compilateur de savoir que cette méthode est contenue dans une bibliothèque native.

Il ne doit pas y avoir d'implémentation même un corps vide pour une méthode déclarée native.

Exemple :

```
class TestJNI1 {
public native void afficherBonjour();
static {
System.loadLibrary("mabibjni");
    }

public static void main(String[] args) {
new TestJNI1().afficherBonjour();
```

```
    }
}
```

Pour pouvoir utiliser une méthode native, il faut tout d'abord charger la bibliothèque. Pour réaliser ce chargement, il utiliser la méthode statique loadLibrary() de la classe system. Il faut obligatoirement s'assurer que la bibliothèque est chargée avant le premier appel de la méthode native.

Le plus simple pour assurer ce chargement est de le demander dans un morceau de code d'initialisation statique de la classe.

Exemple :

```
class TestJNI1 {
  public native void afficherBonjour();
  static {
    System.loadLibrary("mabibjni");
  }
}
```

Le nom de la bibliothèque fournie en paramètre doit être indépendant de la plate-forme utilisée : il faut préciser le nom de la bibliothèque sans son extension. Le nom sera automatiquement adapté selon le système d'exploitation sur lequel le code Java est exécuté.

L'utilisation de la méthode native dans le code Java se fait de la même façon qu'une méthode classique.

Exemple :

```
class TestJNI1 {
  public native void afficherBonjour();
  static {
    System.loadLibrary("mabibjni");
  }

  public static void main(String[] args) {
    new TestJNI1().afficherBonjour();
  }
}
```

28.2. La génération du fichier d'en-tête

L'outil javah fourni avec le JDK permet de générer un fichier d'en-tête qui va contenir la définition dans le langage C des fonctions correspondant aux méthodes déclarées native dans le code source Java.

Javah utilise le byte code pour générer le fichier .h. Il faut donc que la classe Java soit préalablement compilée.

La syntaxe est donc : javah -jni nom_fichier_sans_extension

Exemple :

```
D:\java\test\jni>dir
03/12/2003  14:39       <DIR>          .
03/12/2003  14:39       <DIR>          ..
03/12/2003  14:39                  230 TestJNI1.java
            2 fichier(s)              230 octets
            2 Rép(s)   2 200 772 608 octets libres
D:\java\test\jni>javac TestJNI1.java
D:\java\test\jni>javah -jni TestJNI1
D:\java\test\jni>dir
Répertoire de D:\java\test\jni
03/12/2003  14:39       <DIR> .
```

```
03/12/2003  14:39       <DIR> ..
03/12/2003  14:39                  459 TestJNI1.class
03/12/2003  14:39                  399 TestJNI1.h
03/12/2003  14:39                  230 TestJNI1.java
            3 fichier(s) 1 088 octets
            2 Rép(s) 2 198 208 512 octets libres
D:\java\test\jni>
```

Le fichier TestJNI1.h généré est le suivant :

Exemple :

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class TestJNI1 */

#ifndef _Included_TestJNI1
#define _Included_TestJNI1
#ifdef __cplusplus
extern " C " {
#endif
/*
 *      Class:     TestJNI1
 *      Method:    afficherBonjour
 *      Signature: ()V
 */
JNIEXPORT void JNICALL Java_TestJNI1_afficherBonjour
(JNIEnv *, jobject);
#ifdef __cplusplus
}
#endif
#endif
```

Le nom de chaque fonction native respecte le format suivant :
Java_nomPleinementQualifieDelaClasse_NomDeLaMethode

Ce fichier doit être utilisé dans l'implémentation du code de la fonction.

Même si la méthode native est déclarée sans paramètre, il y a toujours deux paramètres passés à la fonction native :

- un pointeur vers une structure JniEnv : cet objet permet d'utiliser certaines fonctions permettant d'utiliser certains paramètres non primitifs fournis à la méthode
- jobject qui est l'objet lui même : c'est l'équivalent du mot clé this dans le code Java

28.3. L'écriture du code natif en C

La bibliothèque contenant la ou les fonctions qui seront appelées doit être écrite dans un langage (c ou c++) et compilée.

Pour l'écriture en C, facilitée par la génération du fichier.h, il est nécessaire en plus des includes liées au code des fonctions d'inclure deux fichiers d'en-tête :

- jni.h qui est fourni avec le JDK
- le fichier .h généré par la commande javah

Exemple : TestJNI.c

```
#include <jni.h>
#include <stdio.h>
#include "TestJNI1.h"

JNIEXPORT void JNICALL
Java_TestJNI1_afficherBonjour(JNIEnv *env, jobject obj)
```

```
{
  printf(" Bonjour\n ");
  return;
}
```

Il faut compiler ce fichier source sous la forme d'un fichier objet .o

Exemple : avec MinGW sous Windows

```
D:\java\test\jni>gcc -c -I"C:\j2sdk1.4.2_02\include" -I"C:\j2sdk1.4.2_02\include
\win32" -o TestJNI.o TestJNI.c
```

Il faut ensuite définir un fichier .def qui contient la définition des fonctions exportées par la bibliothèque

Exemple : TestJNI.def

```
EXPORTS
Java_TestJNI1_afficherBonjour
```

Il ne reste plus qu'à générer la dll.

Exemple : TestJNI.def

```
D:\java\test\jni>gcc -shared -o mabibjni.dll TestJNI.o TestJNI.def
Warning: resolving _Java_TestJNI1_afficherBonjour by linking to _Java_TestJNI1_a
fficherBonjour@8
Use-enable-stdcall-fixup to disable these warnings
Use-disable-stdcall-fixup to disable these fixups

D:\java\test\jni>dir
Répertoire de D:\java\test\jni
03/12/2003  16:22       <DIR>          .
03/12/2003  16:22       <DIR>          ..
03/12/2003  16:22               12 017 mabibjni.dll
03/12/2003  15:58                  193 TestJNI.c
03/12/2003  16:20                   40 TestJNI.def
03/12/2003  16:04                  543 TestJNI.o
03/12/2003  14:39                  459 TestJNI1.class
03/12/2003  14:39                  399 TestJNI1.h
03/12/2003  14:39                  230 TestJNI1.java
               9 fichier(s)           14 074 octets
               2 Rép(s)   2 198 392 832 octets libres

D:\java\test\jni>
```

Il ne reste plus qu'à exécuter le code Java dans une machine virtuelle.

Exemple :

```
D:\java\test\jni>java TestJNI1
Bonjour
D:\java\test\jni>
```

Il est intéressant de noter que tant que la signature de la méthode native ne change pas, il est inutile de recompiler la classe Java si la fonction dans la bibliothèque est modifiée et recompilée.

28.4. Le passage de paramètres et le renvoi d'une valeur (type primitif)

Une méthode a quasiment toujours besoin de paramètres et souvent besoin de retourner une valeur.

Cette section va définir et utiliser une méthode native qui ajoute deux entiers et renvoie le résultat de l'addition.

Exemple : le code Java

```
class TestJNI1 {
  public native int ajouter(int a, int b);
    static {
      System.loadLibrary("mabibjni");
    }

  public static void main(String[] args) {
    TestJNI1 maclasse = new TestJNI1();
    System.out.println("2 + 3 = " + maclasse.ajouter(2,3));
  }
}
```

La déclaration de la méthode n'a rien de particulier hormis le modificateur native.

La signature de la fonction dans le fichier .h tient des paramètres.

Exemple :

```
JNIEXPORT jint JNICALL Java_TestJNI1_ajouter
(JNIEnv *, jobject, jint, jint);
```

Les deux paramètres sont ajoutés dans la signature de la fonction avec un type particulier jint, défini avec un typedef dans le fichier jni.h. Il y a d'ailleurs des définitions pour toutes les primitives.

Primitive Java	Type natif
boolean	jboolean
byte	jbyte
char	jchar
double	jdouble
int	jint
float	jfloat
long	jlong
short	jshort
void	void

Il suffit ensuite d'écrire l'implémentation du code natif.

Exemple :

```
#include <jni.h>
#include <stdio.h>
#include "TestJNI2.h"

JNIEXPORT jint JNICALL Java_TestJNI2_ajouter
(JNIEnv *env, jobject obj, jint a, jint b)
{
  return a + b;
}
```

Il faut ensuite compiler le code :

Exemple :

```
D:\java\test\jni>gcc -c -I"C:\j2sdk1.4.2_02\include" -I"C:\j2sdk1.4.2_02\include
\win32" -o TestJNI2.o TestJNI2.c
```

Il faut définir le fichier .def : l'exemple ci dessous va construire une bibliothèque qui va contenir les fonctions natives des deux classes Java précédemment définies.

Exemple :

```
EXPORTS
Java_TestJNI1_afficherBonjour
Java_TestJNI2_ajouter
```

Il suffit de générer la bibliothèque.

Exemple :

```
D:\java\test\jni>gcc -shared -o mabibjni.dll TestJNI.c TestJNI2.c TestJNI.def
Warning: resolving _Java_TestJNI1_afficherBonjour by linking to _Java_TestJNI1_a
fficherBonjour@8
Use-enable-stdcall-fixup to disable these warnings
Use-disable-stdcall-fixup to disable these fixups
Warning: resolving _Java_TestJNI2_ajouter by linking to _Java_TestJNI2_ajouter@1
6
```

Il ne reste plus qu'a exécuter le code Java

Exemple :

```
D:\java\test\jni>java TestJNI2
2 + 3 = 5
```

28.5. Le passage de paramètres et le renvoi d'une valeur (type objet)

Les objets sont passés par référence en utilisant une variable de type jobject. Plusieurs autres types sont prédéfinis par JNI pour des objets fréquemment utilisés :

Objet C	Objet Java
jobject	java.lang.object
jstring	java.lang.String
jclass	java.lang.Class
jthrowable	java.lang.Throwable
jarray	type de base pour les tableaux
jintArray	int[]
jlongArray	long[]
jfloatArray	float[]

jdoubleArray	double[]
jobjectArray	Object[]
jbooleanArray	boolean[]
jbyteArray	byte[]
jcharArray	char[]
jshortArray	short[]

Exemple : concaténation de deux chaînes de caractères

```
class TestJNI3 {
  public native String concat(String a, String b);

  static {
    System.loadLibrary("mabibjni");
  }

  public static void main(String[] args) {
    TestJNI3 maclasse = new TestJNI3();
    System.out.println("abc + cde = " + maclasse.concat("abc","cde"));
  }
}
```

La déclaration de la fonction native dans le fichier TestJNI3.h est la suivante :

Exemple :

```
/*
 * Class:     TestJNI3
 * Method:    concat
 * Signature: (Ljava/lang/String;Ljava/lang/String;)Ljava/lang/String;
 */
JNIEXPORT jstring JNICALL Java_TestJNI3_concat
  (JNIEnv *, jobject, jstring, jstring);
```

Pour utiliser les paramètres de type jstring dans le code natif, il faut les transformer en utilisant des fonctions proposées par l'interface de type JNIEnv car le type String de Java n'est pas directement compatible avec les chaînes de caractères C (char *). Il existe des fonctions pour transformer des chaînes codées en UTF-8 ou en Unicode.

Les méthodes pour traiter les chaînes au format UTF-8 sont :

- la méthode GetStringUTFChars() permet de convertir une chaîne de caractères Java en une chaîne de caractères de type C.
- la méthode NewStringUTF() permet de demander la création d'une nouvelle chaîne de caractères.
- la méthode GetStringUTFLength() permet de connaître la taille de la chaîne de caractères.
- la méthode ReleaseStringUTFChars() permet de demander la libération des ressources allouées pour la chaîne de caractères dès que celle ci n'est plus utilisée. Son utilisation permet d'éviter des fuites mémoire.

Les méthodes équivalentes pour les chaines de caractères au format unicode sont : GetStringChars(), NewString(), GetStringUTFLength() et ReleaseStringChars()

Exemple : TestJNI3.c

```
#include <jni.h>
#include <stdio.h>
#include "TestJNI3.h"
JNIEXPORT jstring JNICALL Java_TestJNI3_concat
  (JNIEnv *env, jobject obj, jstring chaine1, jstring chaine2){
    char resultat[256];
    const char *str1 = (*env)->GetStringUTFChars(env, chaine1, 0);
```

```
    const char *str2 = (*env)->GetStringUTFChars(env, chaine2, 0);
    sprintf(resultat,"%s%s", str1, str2);
    (*env)->ReleaseStringUTFChars(env, chaine1, str1);
    (*env)->ReleaseStringUTFChars(env, chaine2, str2);
    return (*env)->NewStringUTF(env, resultat);
}
```

Attention : ce code est très simpliste car il ne vérifie pas un éventuel débordement du tableau résultat.

Après la compilation des différents éléments, l'exécution affiche le résultat escompté.

Exemple :

```
D:\java\test\jni>java TestJNI3
abc + cde = abccde
```

29. JNDI (Java Naming and Directory Interface)

JNDI est l'acronyme de Java Naming and Directory Interface. Cette API fournit une interface unique pour utiliser différents services de nommages ou d'annuaires et définit une API standard pour permettre l'accès à ces services.

Il existe plusieurs types de service de nommage parmi lesquels :

- DNS (Domain Name System) : service de nommage utilisé sur internet pour permettre la correspondance entre un nom de domaine et une adresse IP
- LDAP(Lightweight Directory Access Protocol) : annuaire
- NIS (Network Information System) : service de nommage réseau développé par Sun Microsystems
- COS Naming (Common Object Services) : service de nommage utilisé par Corba pour stocker et obtenir des références sur des objets Corba
- etc, ...

Un service de nommage permet d'associer un nom unique à un objet et faciliter ainsi l'obtention de cet objet.

Un annuaire est un service de nommage qui possède en plus une représentation hiérarchique des objets qu'il contient et un mécanisme de recherche.

JNDI propose donc une abstraction pour permettre l'accès à ces différents services de manière standard. Ceci est possible grâce à l'implémentation de pilotes qui mettent en oeuvre la partie SPI de l'API JNDI. Cette implémentation se charge d'assurer le dialogue entre l'API et le service utilisé.

JNDI possède un rôle particulier dans les architectures applicatives développées en Java car elle est utilisée dans les spécifications de plusieurs API majeures : JDBC, EJB, JMS, ...

De plus, la centralisation de données dans une source unique pour une ou plusieurs applications facilite l'administration de ces données et leur accès.

Pour plus d'informations sur JNDI : http://java.sun.com/products/jndi

Sun propose un excellent tutorial sur JNDI à l'url : http://java.sun.com/products/jndi/tutorial/ .

Pour utiliser JNDI, il faut un service de nommage correctement installé et configuré et un pilote dédié à ce service.

29.1. La présentation de JNDI

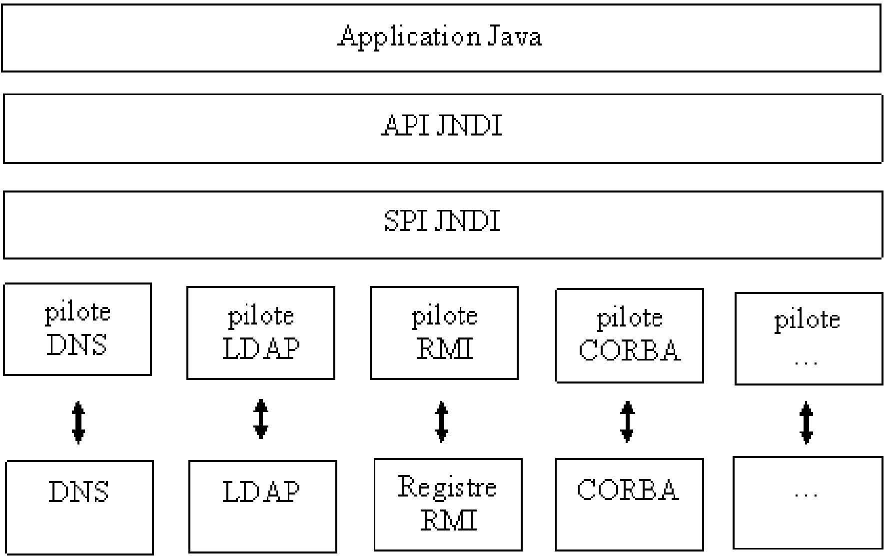
JNDI est composée de deux parties

- Une API utilisée pour le développement des applications
- Une SPI utilisée par les fournisseurs d'une implémentation d'un pilote

Un pilote est un ensemble de classes qui implémentent les interfaces de JNDI pour permettre les interactions avec un service particulier. Ce mode de fonctionnement est identique à celle proposée par l'API JDBC.

Il est donc nécessaire de disposer d'un pilote pour assurer le dialogue entre l'application via l'API et le service de

nommage ou l'annuaire. La partie API est incluse dans le JDK et Sun propose une implémentation des pilotes pour LDAP, DNS et Corba. Pour utiliser d'autres services ou d'autres implémentations il faut utiliser des implémentations des pilotes fournis par des fournisseurs tiers.



Pour définir une connexion, JNDI à besoin d'au moins deux éléments :

- La fabrique du contexte racine : c'est cet objet qui assure le dialogue avec le service utilisé en utilisant le protocole adéquat
- L'url du service à utiliser

JNDI n'est pas utilisable uniquement pour des applications J2EE. Une application standalone peut par exemple réaliser une authentification à partir d'un annuaire via le protocole LDAP.

Ainsi JNDI est inclus dans J2SE depuis la version 1.3 du J2SE. Pour les versions antérieures (J2SE 1.1 et 1.2), il est nécessaire de télécharger JNDI en tant qu'extension standard et de l'installer.

Pilote	J2SE 1.3	J2SE 1.4
LDAP	Oui	Oui
Corba COS	Oui	Oui
Registre RMI	Oui	Oui
DNS	Non	Oui

Il est aussi possible d'utiliser d'autres pilotes fournis séparément par Sun ou par d'autres fournisseurs.

Sun propose une liste des pilotes existant à l'url :

http://java.sun.com/products/jndi/serviceproviders.html

Sun propose aussi le "JNDI/LDAP Booster Pack" qui propose des pilotes pour des serveurs LDAP et un pilote permettant la mise en oeuvre de DSML (Directory Services Markup Language) dont le but est d'accéder à un annuaire avec XML.

29.1.1. Les services de nommage

Il existe de nombreux services de nommage : les plus connus sont sûrement les systèmes de fichiers (File system), les DNS, les annuaires LDAP, ...

Un service de nommage permet d'associer un nom à un objet ou à une référence sur un objet. L'objet associé dépend du service : un fichier dans un système de fichiers, une adresse I.P. dans un DNS, ...

Le nom associé à un objet respect une convention de nommage particulière à chaque type de service.

- Avec un système de fichiers de type Unix, le nom est composé d'éléments séparés par un caractère "/"
- Avec un système de fichiers de type Windows, le nom est composé d'éléments séparés par un caractère "\"
- Avec un service de type DNS, le nom est composé d'éléments séparés par un caractère "." (exemple : www.test.fr).
- Avec un service de type LDAP, le nom désigné par le terme Distinguished Name est composé d'élément séparé par un caractère ",". Un élément est de la forme clé=valeur.

Pour permettre une abstraction des différents formats de noms utilisés par les différents services, JNDI utilise la classe Name.

29.1.2. Les annuaires

Un annuaire est un outil qui permet de stocker et de consulter des informations selon un protocole particulier. Un annuaire est plus particulièrement dédié à la recherche et la lecture d'informations : il est optimisé pour ce type d'activité mais il doit aussi être capable d'ajouter et de modifier des informations.

Les annuaires sont des extensions des services de nommage en ajoutant en plus la possibilité d'associer d'éventuels attributs à chaque objet.

Caractéristiques	Annuaire	Bases de données
Accès aux données	Lecture privilégiée	Lecture et modification
Représentation des données	Hiérarchique	Ensembliste

Les annuaires les plus connus dans le monde réel sont les pages jaunes et les pages blanches du principal opérateur téléphonique. Même si le but de ces deux annuaires est identique (obtenir un numéro de téléphone), la structure des données est différentes :

- Pages blanches : regroupement par département, ville, nom/prenom
- Pages jaunes : regroupement par activités, ville, nom

Les systèmes de fichiers sont aussi des annuaires : ils associent un nom à un fichier mais stockent aussi des attributs liés à ces fichiers (droits d'accès, dates de création et de modification, ...)

29.1.3. Le contexte

Un service de nommage permet d'associer un nom à un objet. Cette association est nommée binding. Un ensemble d'associations nom/objet est nommé un contexte.

Ce contexte est utilisé lors de l'accès à un élément contenu dans le service.

Il existe deux types de contexte :

- Contexte racine
- Sous contexte

Un sous-contexte est un contexte relatif à un contexte racine.

Par exemple, c:\ est un contexte racine dans un système de fichiers de type Windows. Le répertoire windows est un sous contexte du contexte racine (C:\windows) qui est dans ce cas nommé sous répertoire.

Dans DNS, com est un contexte racine et test est un sous contexte (test.com)

29.2. La mise en oeuvre de l'API JNDI

L'API JNDI est contenue dans cinq packages :

Packages	Rôle
javax.naming	Classes et interfaces pour utiliser un service nommage
javax.naming.directory	Etend les fonctionnalités du package javax.naming pour l'utilisation des services de type annuaire
javax.naming.event	Classes et interfaces pour la gestion des événements lors d'un accès à un service
javax.naming.ldap	Etend les fonctionnalités du package javax.naming.directory pour l'utilisation de la version 3 de LDAP
javax.naming.spi	Classes et interfaces dédiées aux Service Provider pour le développement de pilotes

29.2.1. L'interface Name

Cette interface encapsule un nom en permettant de faire abstraction des conventions de nommage utilisées par le service.

Deux classes implémentent cette interface :

- CompositeName : chaque élément qui compose le CompositeName est séparé par un caractère /
- CompoundName : chaque élément issu de la hiérarchie compose le nom selon certaines règles dépendantes de l'implémentation

29.2.2. L'interface Context et la classe IntitialContext

L'interface javax.Naming.Context représente un ensemble de correspondances nom/objet d'un service de nommage. Elle propose des méthodes pour interroger et mettre à jour ces correspondances.

Méthode	Rôle
void bind(String, Object)	Ajouter une nouvelle correspondance entre le nom et l'objet passé en paramètre
void rebind(String, Object)	
Object lookup(String)	Renvoie un objet à partir de son nom
void unbind(String)	Supprimer la correspondance désignée par le nom fourni en paramètre
void rename(String, String)	Modifier le nom d'une correspondance
NamingEnumeration listBindings(String)	Renvoie les objets associés à la correspondance dont le nom est fourni en paramètre
NamingEnumeration list(String)	

Toutes ces méthodes possèdent une version surchargée qui attend le nom de la correspondance sous la forme d'un objet de type Name.

La classe InitialContext qui implémente l'interface Context encapsule le contexte racine : c'est le noeud qui sert de point d'entrée lors de la connexion avec le service.

Toutes les opérations réalisées avec JNDI sont relatives à ce contexte racine.

La classe javax.Naming.InitialContext qui implémente l'interface Context encapsule le point d'entrée dans le service de nommage.

Pour obtenir une instance de la classe InitialContext et ainsi réaliser la connexion au service, plusieurs paramètres sont nécessaires :

- java.naming.factory.initial permet de préciser le nom de la fabrique proposée par le fournisseur. Cette fabrique est en charge de l'instanciation d'un objet de type InitialContext
- java.naming.provider.url : URL du context racine

Plusieurs fabriques sont fournies en standard dans J2SE 1.4 :

Service	Fabrique
CORBA	com.sun.jndi.cosnaming.CNCtxFactory
DNS	com.sun.jndi.dns.DnsContextFactory
LDAP	com.sun.jndi.ldap.LdapCtxFactory
RMI	com.sun.jndi.rmi.registry.RegistryContextFactory

Ces deux paramètres sont obligatoires mais d'autres peuvent être nécessaire notamment ceux concernant la sécurité pour l'accès au service.

L'interface Context définit des constantes pour le nom de ces paramètres.

Il y a plusieurs moyens pour définir ces paramètres :

- les définir sous la forme de variables d'environnement passées à la JVM en utilisant l'option -D
- les définir sous la forme d'une collection de type Hashtable passée en paramètre au constructeur de la classe InitialContext
- les définir dans un fichier nommé jndi.properties accessible dans le classpath

Exemple :

```
Hashtable hashtableEnvironment = new Hashtable();
hashtableEnvironment.put(Context.INITIAL_CONTEXT_FACTORY,
   "com.sun.jndi.fscontext.RefFSContextFactory");
hashtableEnvironment.put(Context.PROVIDER_URL, "file:c:/");
Context context = new InitialContext(hashtableEnvironment);
```

Il est possible de réaliser des opérations particulières à partir du Context. Attention toutefois, toutes ces opérations ne sont pas utilisables avec tous les pilotes. Par exemple, l'accès à un service de type DNS n'est possible qu'en consultation.

29.3. L'utilisation d'un service de nommage

Pour pouvoir utiliser un service de nommage, il faut tout d'abord obtenir un contexte racine qui va encapsuler la connexion au service.

A partir de ce contexte, il est possible de réaliser plusieurs opérations :

- bind : associer un objet avec un nom
- rebind : modifier une association
- unbind : supprimer une association
- lookup : obtenir un objet à partir de son nom
- list : obtenir une liste des associations

Toutes les opérations possèdent deux versions surchargées attendant respectivement :

* Un objet de type Name : cet objet encapsule une séquence ordonnée de un ou plusieurs éléments (l'intérêt de cette classe est de permettre la manipulation individuel de chaque élément).
* Une chaine de caractères : elle contient la séquence

29.3.1. L'obtention d'un objet

Pour obtenir un objet du service de nommage, utiliser la méthode lookup() du contexte.

Exemple :

```
import javax.naming.*;
...
  public String getValeur() throws NamingException {
    Context context = new InitialContext();
    return (String) context.lookup("/config/monApplication");
  }
```

Ceci peut permettre de facilement stocker des options de configuration d'une application, plutôt que de les stocker dans un fichier de configuration. Ceci est d'autant plus intéressant si le service qui stocke ces données est accessible via le réseau car cela permet de centraliser ces options de configuration.

Il peut permettre aussi de stocker des données "sensibles" comme des noms d'utilisateur et des mots de passe pour accéder à une ressource et ainsi empêcher leur accès en clair dans un fichier de configuration.

29.3.2. Le stockage d'un objet

Généralement les objets à stocker doivent être d'un type particulier, dépendant du pilote utilisé : il est fréquent que de tels objets doivent implémenter une interface (java.io.Serializable, java.rmi.Remote, etc ...)

La méthode bind() permet d'associer un objet à un nom.

Exemple :

```
import javax.naming.*;
...
  public void createName() throws NamingException {
    Context context = new InitialContext();
    context.bind("/config/monApplication", "valeur");
  }
```

29.4. L'utilisation avec un DNS

A partir de J2SE 1.4, Sun propose en standard une implémentation permettant d'accéder à un DNS via JNDI.

Exemple :

```
import javax.naming.*;
import javax.naming.directory.*;
import java.util.*;

public class TestDNS2 {

     public static void main(String[] args) {
        try {
```

```
            Hashtable env = new Hashtable();
            env.put("java.naming.factory.initial",
                  "com.sun.jndi.dns.DnsContextFactory");
            env.put("java.naming.provider.url", "dns://80.10.246.2/");

            DirContext ctx = new InitialDirContext(env);
            Attributes attrs = ctx.getAttributes("java.sun.com",
                  new String[] { "A" });

            for (NamingEnumeration ae = attrs.getAll(); ae.hasMoreElements();) {
               Attribute attr = (Attribute) ae.next();
               String attrId = attr.getID();
               for (Enumeration vals = attr.getAll();
                               vals.hasMoreElements();
                               System.out.println(attrId + ": " + vals.nextElement())
                            );
            }
            ctx.close();
         } catch (Exception e) {
            System.err.println("Probleme lors de l'interrogation du DNS: " + e);
            e.printStackTrace();
         }
      }
}
```

Pour permettre une exécution correcte de ce programme, il est nécessaire de mettre l'adresse IP du serveur DNS utilisé.

Lors de l'exécution, il faut fournir en paramètre le nom d'un domaine et d'un serveur.

29.5. L'utilisation du File System Context Provider de Sun

C'est une implémentation de référence proposée par Sun qui permet un accès à un système de fichier via JNDI.

Cela peut paraître étonnant mais un système de fichier peut être vu comme un service de nommage qui associe un nom (par exemple c:\temp\test.txt) à un fichier ou un répertoire

Cette implémentation n'est pas fournie en standard avec le JDK mais elle peut être téléchargée à l'url http://java.sun.com/products/jndi/downloads/index.html

La version utilisée dans le cette section est la 1_2 beta3. Il suffit de décompresser le fichier fscontext-1_2-beta3.zip dans un répertoire du système et d'ajouter les fichiers fscontext.jar et providerutil.jar du sous répertoire lib décompressé dans le classpath de l'application.

Exemple : obtenir la liste de tous les fichiers et répertoires à la racine du disque C:

```
import java.util.Hashtable;
import javax.naming.Binding;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingEnumeration;
import javax.naming.NamingException;

public class TestJNDI {

  public static void main(String[] args) {

    try {
      Hashtable hashtableEnvironment = new Hashtable();
      hashtableEnvironment.put(Context.INITIAL_CONTEXT_FACTORY,
            "com.sun.jndi.fscontext.RefFSContextFactory");
      hashtableEnvironment.put(Context.PROVIDER_URL, "file:c:/");

      Context context = new InitialContext(hashtableEnvironment);
      NamingEnumeration namingEnumeration = context.listBindings("");
```

```
      while (namingEnumeration.hasMore()) {
        Binding binding = (Binding) namingEnumeration.next();
        System.out.println(binding.getName());
      }

      context.close();
    } catch (NamingException namingexception) {
      namingexception.printStackTrace();
    }
  }
}
```

Il est aussi possible de rechercher un fichier dans un répertoire. Dans ce cas, le contexte initial précisé est le répertoire dans lequel le fichier doit être recherché. La méthode lookup() recherche uniquement dans ce répertoire

Exemple :

```
import java.io.File;
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class TestJNDI2 {

  public static void main(String argv[]) {
    Hashtable env = new Hashtable();
    env.put(Context.INITIAL_CONTEXT_FACTORY,
          "com.sun.jndi.fscontext.RefFSContextFactory");
    env.put(Context.PROVIDER_URL, "file:c:/");

    try {
      Context ctx = new InitialContext(env);
      File fichier = (File) ctx.lookup("boot.ini");
      System.out.println("objet trouve = " + fichier);
    } catch (NamingException e) {
      e.printStackTrace();
    }
  }
}
```

Attention, le cast vers la classe effectué sur l'objet retourné par la méthode lookup() doit être pertinent en fonction du contexte.

29.6. LDAP

LDAP, acronyme de Lightweight Directory Access Protocol, est un protocole de communication vers un annuaire en utilisant TCP/IP. Il est une simplification du protocole X 500 (d'ou le L de Lightweight).

Le but principal est de retrouver des données insérées dans l'annuaire. Ce protocole est donc optimisé pour la lecture et la recherche d'informations.

LDAP est un protocole largement supporté par l'industrie informatique : il existe de nombreuses implémentations libres et commerciales : Microsoft Active Directory, OpenLDAP, Netscape Directory Server, Sun NIS, Novell NDS, ..

Ce protocole ne précise pas comment ces données sont stockées sur le serveur. Ainsi un serveur de type LDAP peut stocker n'importe qu'elle type de données : ce sont souvent des ressources (personnes, matériels réseaux, ...).

La version actuelle de LDAP est la v3 définie par les RFC 2252 et RFR 2256 de l'IETF.

Dans un annuaire LDAP, les noeuds sont organisés sous une forme arborescente hiérarchique nommée le DIT (Direct Information Tree). Chaque noeud de cette arborescence représente une entrée dans l'annuaire. Chaque entrée contient un

objet qui possède un ou plusieurs attributs dont les valeurs permettent d'obtenir des informations sur l'objet. Un objet appartient à une classe au sens LDAP.

La première et unique entrée dans l'arborescence est nommée racine.

Chaque objet possède un Relatif Distinguish Name (RDN) qui correspond à une paire clé/valeur d'un attribut obligatoire. Un objet est identifié de façon unique grâce à référence unique dans le DIT : son Distinguish Name (DN) qui est composé de l'ensemble des RDN de chaque objet père dans l'arborescence lu de droite à gauche et son RDN (ceci correspond donc au DN de l'entrée père et de son RDN). Cette référence représente donc le chemin d'accès depuis la racine de l'arborescence. Le DN se lit de droite à gauche puisque la racine est à droite.

La convention de nommage utilisée pour le DN, utilise la virgule comme séparateur et se lit de droite à gauche.

Exemple :

```
uid=jm,ou=utilisateur,o=test.com
```

Le premier élément du DN, nommé Relative Distinguished Name (RDN), est composé d'une paire clé/valeur. Comme valeur de clé, LDAP utilise généralement un mnémonique :

Mnnémonique	Libellé	Description
dn	Distinguished name	Nom unique dans l'arborescence
uid	Userid	Identifiant unique pour l'utilisateur
cn	Common name	Nom et prénom d'un utilisateur
givenname	First name	Prénom d'un utilisateur
sn	Surname	Nom de l'utilisateur
l	Location	Ville de l'utilisateur
o	Organization	Généralement la racine de l'annuaire (exemple : le nom de l'entreprise)
ou	Organizational unit	Généralement une branche de l'arbre (exemple : une division, un département ou un service)
st	State	Etat du pays de l'utilisateur
c	Country	pays de l'utilisateur
Mail	Email	Email de l'utilisateur

Un élément qui compose une entrée dans l'annuaire est nommé objet. Chaque objet peut contenir des attributs obligatoires ou facultatifs. Un attribut correspond à une propriété d'un objet, par exemple un email ou un numéro de téléphone pour une personne. Un attribut se présente sous la forme d'une paire clé/valeur(s).

Les classes caractérisent les objets en définissant les attributs optionnels et obligatoires qui les composent. Il existe des attributs standards communément utilisés mais il est aussi possible d'en définir d'autre.

L'ensemble des règles qui définissent l'arborescence et les attributs utilisables sont stockés dans un schéma. : ce dernier permet donc de définir les classes et les objets pouvant être stockées dans l'annuaire. Un annuaire pour supporter plusieurs schémas.

Une fonctionnalité intéressante est la possibilité de pouvoir stocker des objets Java directement dans l'annuaire et de pouvoir les retrouver en utilisant le protocole LDAP. Ces objets peuvent avoir des fonctionnalités diverses telle qu'une connexion à source de données, un objet contenant des options de paramétrage de l'application, etc ...

Un serveur LDAP propose les fonctionnalités de base suivantes :

* Connexion/déconnexion au serveur
* Gestion de la sécurité lors d'accès aux objets
* Ajout, modification, suppression d'objets
* Gestion d'attributs sur les objets
* Recherche d'objets

29.6.1. L'outil OpenLDAP

Il faut télécharger et exécuter le fichier openldap-2.2.29-db-4.3.29-openssl-0.9.8a-win32_Setup.exe à l'url :
http://download.bergmans.us/openldap/openldap-2.2.29/openldap-2.2.29-db-4.3.29-openssl-0.9.8a-win32_Setup.exe

Il faut sélectionner la langue d'installation entre anglais et allemand.

Un assistant guide l'utilisateur dans les différentes étapes de l'installation :

* sur la page d'accueil : cliquez sur le bouton « Next »
* sur la page « Licence Agreement » : lisez la licence et si vous l'acceptez cliquez sur le bouton radio « I accept the agreement » et cliquez sur le bouton « Next »
* sur la page « Select Destination Location », sélectionnez le répertoire de destination et cliquez sur le bouton « Next ». Il est préférable de choisir un répertoire sans espace (exemple : C:\OpenLDAP) plutôt que le répertoire C:\Program Files\OpenLDAP proposé par défaut
* sur la page « Select Components » : laissez la sélection par défaut et cliquez sur le bouton « Next »
* sur la page « Select Start Menu Folder », cliquez sur le bouton « Next »
* sur la page « Select Additional tasks », cliquez sur le bouton « Next »
* sur la page « Ready to Install », cliquez sur le bouton « Install »
* Les fichiers sont copiés sur le système d'exploitation
* sur la page « Completing the OpenLDAP Setup Wizard », cliquez sur le bouton « Finish »

OpenLDAP propose en standard plusieurs schémas prédéfinis stockés dans le sous répertoire schema.

Le fichier slapd.conf contient les principaux paramètres. Il est installé pré-paramétré lors de l'installation dans le répertoire d'installation d'OpenLDAP (c:\openldap dans cette section).

Au début du fichier, il faut ajouter le schéma java pour utiliser openldap avec JNDI pour stocker des objets java.

Exemple :

```
#
ucdata-path       ./ucdata
include           ./schema/core.schema
include>/b<>b<           ./schema/java.schema>/b<
...
```

Il faut ensuite configurer la base de données, le suffixe qui est la racine du serveur et le compte de l'administrateur du serveur (root).

Exemple :

```
#######################################################################
# BDB database definitions
#######################################################################
database              bdb
suffix                "dc=my-domain,dc=com"
rootdn                "cn=Manager,dc=my-domain,dc=com"
# Cleartext passwords, especially for the rootdn, should
# be avoid.  See slappasswd(8) and slapd.conf(5) for details.
# Use of strong authentication encouraged.
rootpw                secret
# The database directory MUST exist prior to running slapd AND
# should only be accessible by the slapd and slap tools.
# Mode 700 recommended.
directory             ./data
```

```
# Indices to maintain
index     objectClass      eq
```

Il faut remplacer la valeur des clés suffix et rootdn par les valeurs appropriés au contexte.

Exemple :

```
suffix                "dc=test-ldap,dc=net"
rootdn              "cn=ldap-admin,dc=test-ldap,dc=net"
```

Pour insérer le mot de passe dans le fichier slapd.conf, il faut le crypter grâce à la commande slappasswd

Exemple :

```
C:\openldap>slappasswd -s ldap-admin
{SSHA}ZUPUkq7mt21rEmrFgFc0cgk9izpwL7oY
```

Il suffit alors de remplacer dans le fichier slapd.conf la ligne

rootpw secret

par la ligne ci dessous qui contient le mot de passe crypté

rootpw {SSHA}ZUPUkq7mt21rEmrFgFc0cgk9izpwL7oY

Pour lancer le serveur LDAP, il suffit de double cliquer sur le fichier slapd.exe

Il ne faut pas fermer cette fenêtre dans laquelle le serveur s'exécute. Pour éviter d'avoir une fenêtre DOS ouverte, il faut utiliser le serveur en tant que service en exécutant la commande net start OpenLDAP-slapd.

Exemple :

```
C:\OpenLDAP>net start OpenLDAP-slapd
Le service OpenLDAP Directory Service démarre..
Le service OpenLDAP Directory Service a démarré.
```

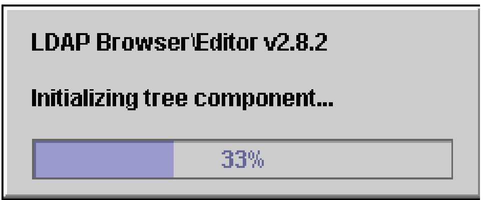
Par défaut, les serveurs de type LDAP utilise le port 389 : c'est le cas pour OpenLDAP.

29.6.2. LDAPBrowser

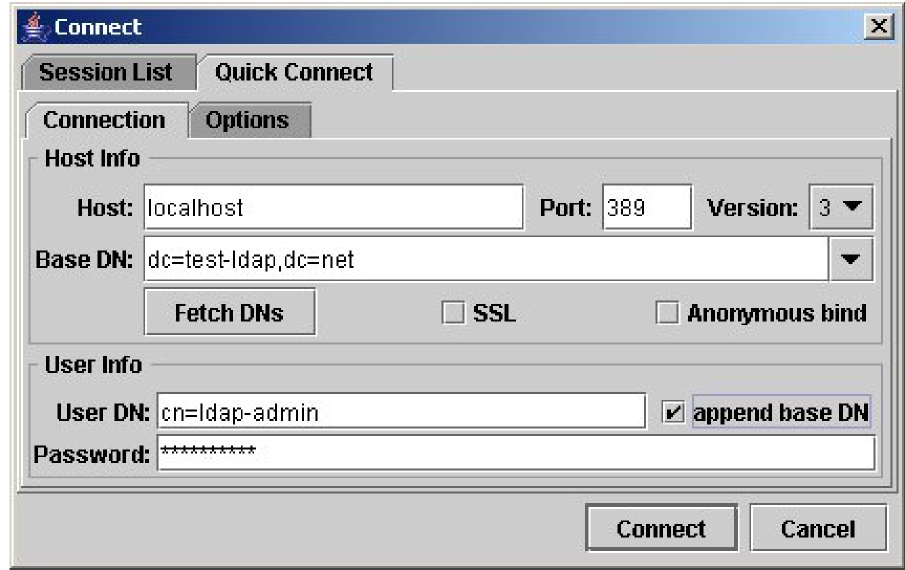
http://www-unix.mcs.anl.gov/~gawor/ldap/

Télécharger le fichier Browser282b2.zip sur la page http://www-unix.mcs.anl.gov/~gawor/ldap/download.html et le décompresser dans un répertoire du système.

Pour lancer l'application, il suffit de double cliquer sur le fichier lbe.bat.

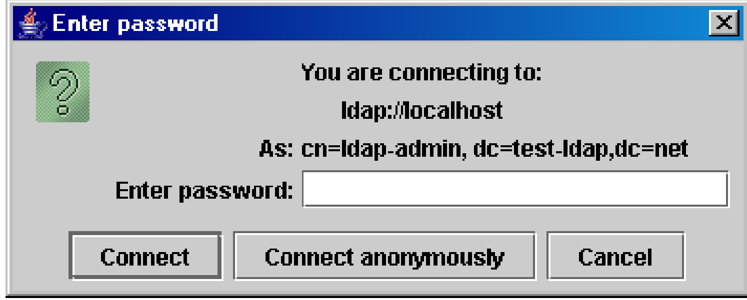


Dans la boîte de dialogue « Connect », sélectionnez l'onglet « Quick Connect » et saisissez les informations nécessaires à la connexion.



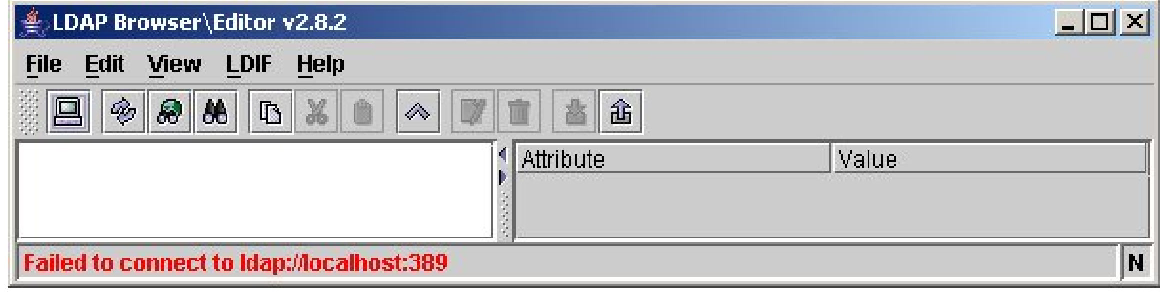
Cliquez sur le bouton « Connect ».

Si le mot de passe n'est pas saisi, une boite de dialogue permet sa saisie

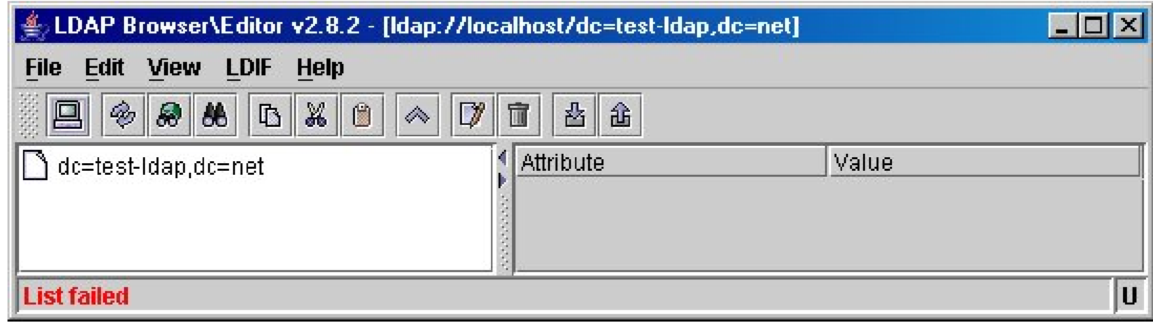


Il suffit alors de saisir le mot de passe défini dans le fichier slapd.conf et de cliquer sur le bouton « Connect ».

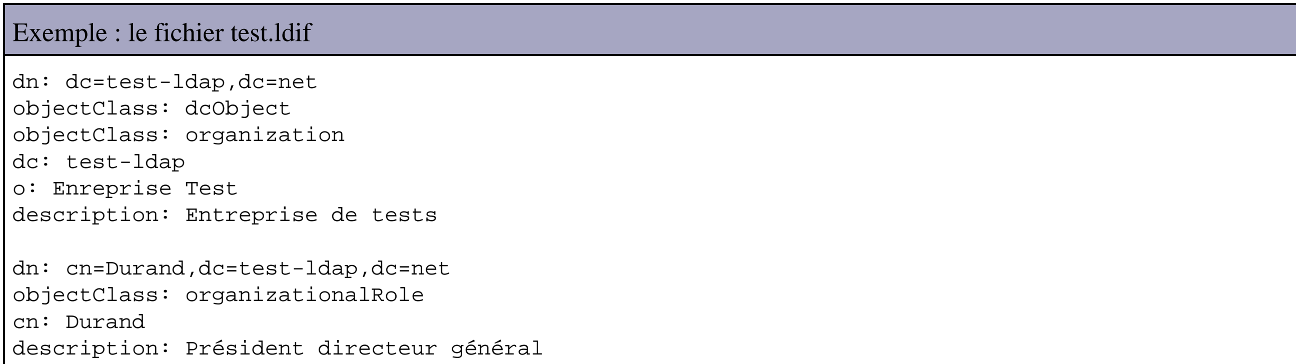
Si les informations saisies ne permettent pas de réussir la connexion, alors le message « Failed to connect » est affiché.



Si l'annuaire est vide, alors le message « List failed » est affiché



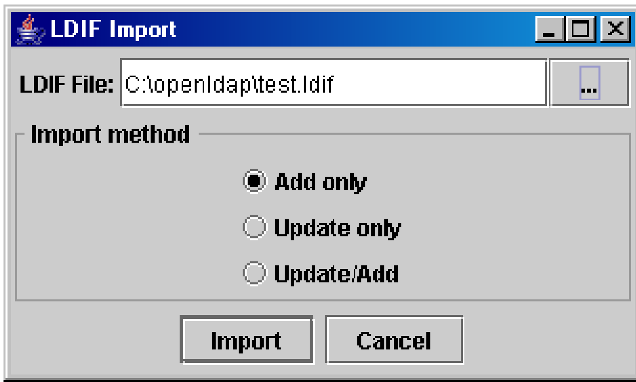
Pour initialiser l'annuaire, le plus facile est d'écrire un fichier au format LDIF (Lightweight Data Interchange Format). Ce format permet d'importer ou d'exporter des données de l'annuaire. Il permet aussi facilement de modifier des données dans l'annuaire. Il est détaillé dans la section suivante.



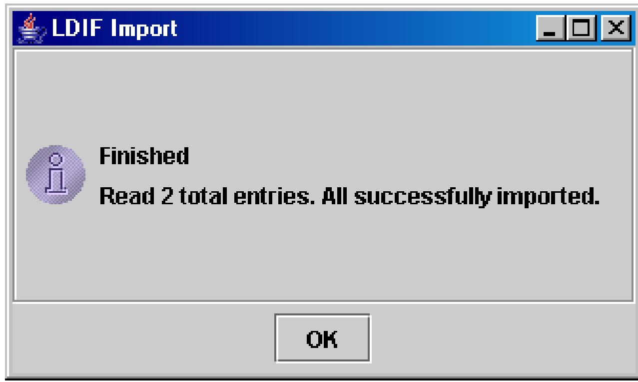
Exemple : le fichier test.ldif

```
dn: dc=test-ldap,dc=net
objectClass: dcObject
objectClass: organization
dc: test-ldap
o: Enreprise Test
description: Entreprise de tests

dn: cn=Durand,dc=test-ldap,dc=net
objectClass: organizationalRole
cn: Durand
description: Président directeur général
```

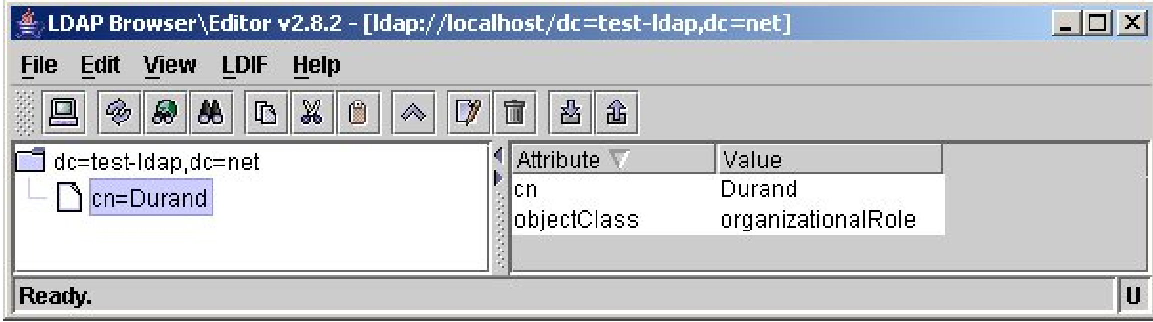
Pour insérer les données du fichier test.ldif, il faut sélectionner la racine et utiliser l'option Import du menu LDIF.



Sélectionner le fichier .ldif et cliquez sur le bouton « Import ».



Les deux entrées sont affichées dans l'arborescence du serveur.



29.6.3. LDIF

Le format LDIF permet de réaliser des opérations d'import/export de données d'un annuaire.

Le format général de ce format est le suivant :

Exemple :

```
[<id>]
dn: <distinguished name>
objectclass: <objectclass>
objectclass: <objectclass>
...
<attribut> : <valeur>
<attribut> : <valeur>
...
```

Chaque entrée est séparée dans le fichier par une ligne vide.

<id> est un entier positif facultatif qui représente un identifiant dans les données du serveur.

Chaque élément définit dans le fichier est séparé par une ligne vide. Il commence par son DN

Chaque attribut est définit sur sa propose ligne. La définition peut se poursuivre sur la ligne suivante si celle-ci commence par une espace ou une tabulation.

Pour fournir plusieurs valeurs à un attribut, il suffit de répéter la clé de cet attribut sur une ligne pour chaque valeur.

Si la valeur d'un attribut contient des caractères non imprimables (des données binaires comme une image par exemple) alors la clé de l'attribut est suivi de :: et la valeur est encodé en base 64.

Le format LDIF permet également d'effectuer des modifications de données grâce à des opérations : add (ajouter une entrée), delete (supprimer une entrée), modrdn (modifier le rdn)

29.7. L'utilisation avec un annuaire LDAP

L'API JNDI permet un accès à un annuaire LDAP.

29.7.1. L'interface DirContext

L'interface DirContext est une classe fille de l'interface Context. Elle propose des fonctionnalités pour utiliser un service de nommage et propose en plus des fonctionnalités dédiées aux annuaires telles que la gestion des attributs et la recherche d'éléments.

Méthode	Rôle
Void bind(String, Object, Attributes)	Permet d'associer un objet avec des attributs à un nom
Void rebind(String, Object , Attributes)	Permet de modifier l'associer d'un objet avec des attributs à un nom
Attributes getAttributes(String)	Permet d'obtenir tous les attributs de l'objet associé au nom fourni en paramètre
Attributes getAttributes(String, String [])	Permet d'obtenir les attributs de l'objet associé au nom fourni en paramètre. Seuls les attributs fournis en paramètres sont retournés par la méthode
Void modifyAttributes(String, int, Attributes)	Permet de mettre à jour des attributs dans un ordre quelconque L'entier permet de préciser le type de mise à jour à effectuer sous la forme de constantes : ADD_ATTRIBUTE, REPLACE_ATTRIBUTE et REMOVE_ATTRIBUTE
Void modifyAttributes(String, ModificationItem [])	Permet de mettre à jour des attributs dans l'ordre des éléments du tableau fourni en paramètre
NamingEnumeration search()	Rechercher des entrées dans l'annuaire selon des critères fournis sous la forme d'un filtre. Il existe plusieurs surcharges de cette méthode
DirContext getSchema(String)	Retourne le schéma associé à un nom

Pour pouvoir accéder à un annuaire, les étapes sont similaires à celles d'un accès à un service de nommage. Il faut obtenir une instance de type DirContext en instanciant un objet de type InitialDirContext(). Cet objet a besoin de paramètres généralement fournis sous la forme d'une collection de type Hashtable.

Ces paramètres sont les mêmes que pour un accès à un service de nommage.

29.7.2. La classe InitialDirContext

L'instanciation d'un objet de type InitialDirContext permet de se connecter à l'annuaire et de se positionner à un endroit précis de l'arborescence de l'annuaire nommé contexte initial.

Toutes les opérations alors réalisées dans l'annuaire le seront relativement à ce contexte initial.

Pour se connecter à un serveur LDAP, il faut obtenir un objet qui implémente l'interface DirContext : c'est généralement un objet de type InitialDirContext qui est obtenu en utilisant une collection de type Hashtable contenant les paramètres de connexion fourni à une fabrique dédiée.

Afin de permettre de réaliser la connexion, il est nécessaire de fournir des paramètres pour configurer son environnement. Ces paramètres sont fournis au constructeur de la classe InitialDirContext sous la forme d'un objet de type Hashtable : ces paramètres concernent plusieurs types d'informations :

- Le fournisseur de l'implémentation
- La localisation de l'annuaire
- La sécurité d'accès

Deux paramètres sont obligatoires :

Context.INITIAL_CONTEXT_FACTORY	permet de préciser la classe fournie par le fournisseur
Context.PROVIDER_URL	permet de préciser une url pour localiser l'annuaire. Le format de cette url dépend du fournisseur

Exemple :

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.ldap.LdapCtxFactory");
env.put(Context.PROVIDER_URL,"ldap://localhost:389");
DirContext dircontext = new InitialDirContext(env);
```

Si l'accès au serveur est sécurisé, il faut fournir des paramètres supplémentaires pour permettre cette authentification : le type de sécurité utilisée, le DN d'un utilisateur et son mot de passe :

Context.SECURITY_AUTHENTICATION	Permet de préciser le type de sécurité utilisé. Les valeurs possibles sont : simple, SSL, SASL
Context.SECURITY_PRINCIPAL	Permet de préciser le Distinguished Name de l'utilisateur
Context.SECURITY_CREDENTIALS	Le mot de passe de l'utilisateur

LDAP supporte trois modes de sécurité :

- ***Simple:*** pas de cryptage du DN de l'utilisateur ni de son mot de passe
- ***SSL:*** utilisation du cryptage SSL à travers le réseau si le serveur LDAP le supporte
- ***SASL:*** utilisation des algorithmes MD5/Kerberos

Exemple :

```
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.NamingException;
import javax.naming.directory.DirContext;
import javax.naming.directory.InitialDirContext;

public class TestLDAP {

  public static void main(String[] args) {
    Hashtable env = new Hashtable();
    env
        .put(Context.INITIAL_CONTEXT_FACTORY,
            "com.sun.jndi.ldap.LdapCtxFactory");
    env.put(Context.PROVIDER_URL, "ldap://localhost:389");
    env.put(Context.SECURITY_AUTHENTICATION, "simple");
    env.put(Context.SECURITY_PRINCIPAL, "cn=ldap-admin,dc=test-ldap,dc=net");
    env.put(Context.SECURITY_CREDENTIALS, "inconnu");

    DirContext dirContext;

    try {
      dirContext = new InitialDirContext(env);
      dirContext.close();
    } catch (NamingException e) {
      System.err.println("Erreur lors de l'acces au serveur LDAP" + e);
      e.printStackTrace();
    }
  }
}
```

Comme l'objet InitialDircontext encapsule la connexion vers l'annuaire, il est nécessaire de fermer cette connexion dès que celle ci n'est plus utilisée en faisant appel à la méthode close().

La plupart des méthodes de la classe InitialDirContext peuvent lever une exception de type NamingException.

Si les informations de connexion au serveur sont erronées alors une exception de type javax.naming.CommunicationException

Si les informations fournie pour l'authentification sont erronées alors une exception de type javax.naming.AuthenticationException est levée avec le message «[LDAP: error code 49 - Invalid Credentials]»

A partir d'une instance de DirContext, il est possible d'accéder et de réaliser des opérations dans l'annuaire.

29.7.3. Les attributs

Pour manipuler les attributs d'un objet, deux interfaces existent :

- Attributes : qui encapsulent les différents attributs d'un objet
- Attribut qui encapsule la valeur d'un attribut

Exemple :

```
dirContext = new InitialDirContext(env);
Attributes attributs = dirContext.getAttributes("cn=Dupont,dc=test-ldap,dc=net");
Attributs attribut = (Attribut) attributs.get("description") ;
System.out.println("Description : " + attribut.get());
```

Deux classes implémentent respectivement ces deux interfaces : BasicAttributes et BasicAttribut

Il est possible d'instancier une liste d'attributs par exemple pour les associer à un nouvel objet ajoutés dans l'annuaire.

Exemple :

```
Attributes attributes = new BasicAttributes(true);
Attribute attribut = new BasicAttribute("telephoneNumber");
attribut.add("99.99.99.99.99");
attributes.put(attribut);
```

29.7.4. L'utilisation d'objets Java

La possibilité de stocker des objets Java dans un annuaire LDAP offre plusieurs intérêts :

- Stocker des objets accessibles par plusieurs applications
- Stocker des objets entre plusieurs exécutions d'une même application
- Stocker des objets pour échanger des données entre plusieurs applications

A partir d'un objet de type contexte, il suffit de faire appel à la méthode bind() qui attend en paramètre un nom d'objet et un objet. Cette méthode va ajouter une entrée dans l'annuaire qui va associer le nom de l'objet à l'objet fourni en paramètre.

La méthode lookup() d'un objet de type context permet d'obtenir un objet Java stockée dans l'annuaire à partir de son nom.

Ces deux méthodes peuvent lever une exception de type NamingException lors de leur exécution.

29.7.5. Le stockage d'objets Java

La plupart des annuaires permettent le stockage d'objets Java, sous réserve que l'annuaire le propose et que le schéma adéquat soit utilisé dans la configuration du serveur, ce qui n'est généralement pas le cas par défaut.

Le stockage se fait en utilisant la méthode bind() du contexte

Exemple :

```
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.NamingException;
import javax.naming.directory.DirContext;
import javax.naming.directory.InitialDirContext;

public class TestLDAP2 {

  public static void main(String[] args) {
    Hashtable env = new Hashtable();
    env
        .put(Context.INITIAL_CONTEXT_FACTORY,
            "com.sun.jndi.ldap.LdapCtxFactory");
    env.put(Context.PROVIDER_URL, "ldap://localhost:389");
    env.put(Context.SECURITY_AUTHENTICATION, "simple");
    env.put(Context.SECURITY_PRINCIPAL, "cn=ldap-admin,dc=test-ldap,dc=net");
    env.put(Context.SECURITY_CREDENTIALS, "ldap-admin");
    DirContext dirContext;

    try {

      dirContext = new InitialDirContext(env);
      MonObjet objet = new MonObjet("valeur1","valeur2");

      dirContext.bind("cn=monobject,dc=test-ldap,dc=net", objet);
      dirContext.close();

    } catch (NamingException e) {
      System.err.println("Erreur lors de l'acces au serveur LDAP" + e);
      e.printStackTrace();
    }
    System.out.println("fin des traitements");
  }
}
```

Les objets Java peuvent être stockés de différentes manières selon le serveur :

- Stockage des objets eux mêmes sous la forme sérialisée
- Stockage d'une référence mémoire vers l'objet Java : cette référence est encapsulée dans un objet de type java.naming.Reference
- Stockage des champs de l'objet sous la forme d'attributs : l'objet ainsi stocké doit obligatoirement implémenter l'interface DirContext.

L'implémentation des toutes ces méthodes est laissée libre au serveur mais au moins une doit être proposée.

Pour le stockage sous la forme sérialisée, il est nécessaire que l'objet stocké implémente l'interface java.io.Serilizable. C'est la solution la plus facile à mettre en oeuvre

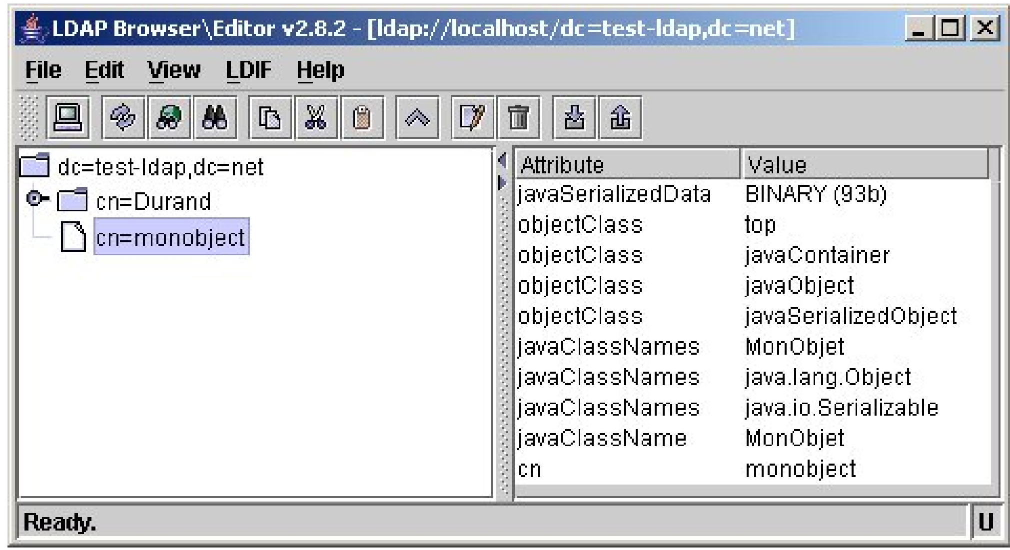
Exemple :

```
import java.io.Serializable;

public class MonObjet implements Serializable {

  private static final long serialVersionUID = 3309572647822157460L;
  private String champ1;
  private String champ2;

  public MonObjet() {
    super();
  }

  public MonObjet(String champ1, String champ2) {
    super();
    this.champ1 = champ1;
    this.champ2 = champ2;
  }
```

```
  public String getChamp1() {
    return champ1;
  }

  public void setChamp1(String champ1) {
    this.champ1 = champ1;
  }

  public String getChamp2() {
    return champ2;
  }

  public void setChamp2(String champ2) {
    this.champ2 = champ2;
  }
}
```

Une exception de type java.lang.IllegalArgumentException est levée si l'objet ne peut pas être ajouté dans l'annuaire si l'objet ne respecte pas les règles pour être ajouté dans l'annuaire. Avec OpenLDAP, cette exception est levée avec le message « can only bind Referenceable, Serializable, DirContext ».

Si tout ce passe bien, l'objet est ajouté dans l'annuaire sous sa forme sérialisée.



29.7.6. L'obtention d'un objet Java

Pour obtenir un objet stocké, il faut utiliser la méthode lookup()

Exemple :

```
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.NamingException;
import javax.naming.directory.DirContext;
import javax.naming.directory.InitialDirContext;

public class TestLDAP3 {

  public static void main(String[] args) {
    Hashtable env = new Hashtable();
    env
        .put(Context.INITIAL_CONTEXT_FACTORY,
            "com.sun.jndi.ldap.LdapCtxFactory");
    env.put(Context.PROVIDER_URL, "ldap://localhost:389");
    env.put(Context.SECURITY_AUTHENTICATION, "simple");
```

```
    env.put(Context.SECURITY_PRINCIPAL, "cn=ldap-admin,dc=test-ldap,dc=net");
    env.put(Context.SECURITY_CREDENTIALS, "ldap-admin");
    DirContext dirContext;

    try {

      dirContext = new InitialDirContext(env);
      MonObjet objet = (MonObjet) dirContext.lookup("cn=monobject,dc=test-ldap,dc=net");

      System.out.println("champ1="+objet.getChamp1());
      System.out.println("champ2="+objet.getChamp2());

      dirContext.close();
    } catch (NamingException e) {
      System.err.println("Erreur lors de l'acces au serveur LDAP" + e);
      e.printStackTrace();
    }
    System.out.println("fin des traitements");
  }
}
```

Résultat :

```
champ1=valeur1
champ2=valeur2
fin des traitements
```

Si le DN fourni en paramètre de la méthode lookup ne correspond pas à celui d'un objet stocké dans l'annuaire, une exception de type javax.naming.NameNotFoundException avec le message « [LDAP: error code 32 - No Such Object] » est levée.

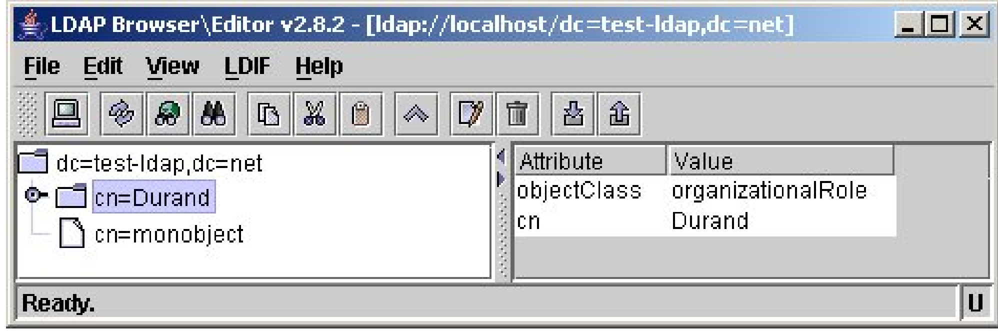
29.7.7. La modification d'un objet

La méthode modifyAttributes() de la classe Dircontext permet de modifier les attributs d'un objet stocké dans l'annuaire. La méthode modifyAttributes() possède plusieurs surcharges.

Différentes opérations sont réalisables avec cette méthode en utilisant des constantes prédéfinies pour chaque type :

- ADD_ATTRIBUTE : ajout d'un attribut
- REMOVE_ATTRIBUTE : suppression d'un attribut
- REPLACE_ATTRIBUTE : modification d'un attribut

Ces modifications sont soumises aux restrictions mises en place sur le serveur au niveau du schéma.

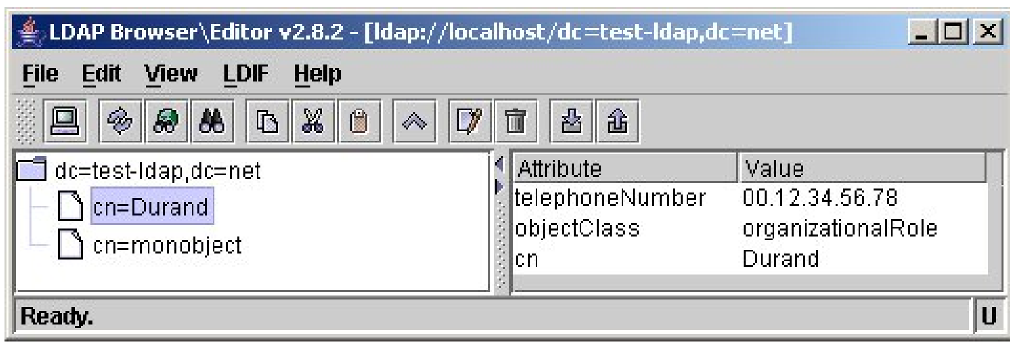


Exemple :

```
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.NamingException;
import javax.naming.directory.Attribute;
import javax.naming.directory.Attributes;
```

```
import javax.naming.directory.BasicAttribute;
import javax.naming.directory.BasicAttributes;
import javax.naming.directory.DirContext;
import javax.naming.directory.InitialDirContext;

public class TestLDAP4 {

  public static void main(String[] args) {
    Hashtable env = new Hashtable();
    env.put(Context.INITIAL_CONTEXT_FACTORY,
          "com.sun.jndi.ldap.LdapCtxFactory");
    env.put(Context.PROVIDER_URL, "ldap://localhost:389");
    env.put(Context.SECURITY_AUTHENTICATION, "simple");
    env.put(Context.SECURITY_PRINCIPAL, "cn=ldap-admin,dc=test-ldap,dc=net");
    env.put(Context.SECURITY_CREDENTIALS, "ldap-admin");
    DirContext dirContext;

    try {

      dirContext = new InitialDirContext(env);

      Attributes attributes = new BasicAttributes(true);
      Attribute attribut = new BasicAttribute("telephoneNumber");
      attribut.add("99.99.99.99.99");
      attributes.put(attribut);

      dirContext.modifyAttributes("cn=Durand,dc=test-ldap,dc=net",
            DirContext.ADD_ATTRIBUTE,attributes);
      dirContext.close();
    } catch (NamingException e) {
      System.err.println("Erreur lors de l'acces au serveur LDAP" + e);
      e.printStackTrace();
    }
    System.out.println("fin des traitements");
  }
}
```

Suite à l'exécution de ce programme, l'attribut est ajouté.



Si l'attribut modifié n'est pas défini dans le schéma alors une exception de type javax.naming.directory.SchemaViolationException avec le message « [LDAP: error code 65 - attribute 'xxx' not allowed] » est levée.

Si l'attribut est ajouté alors qu'il existe déjà, une exception de type javax.naming.directory.AttributeInUseException avec le message « [LDAP: error code 20 - modify/add: xxx: value #0 already exists] » est levée.

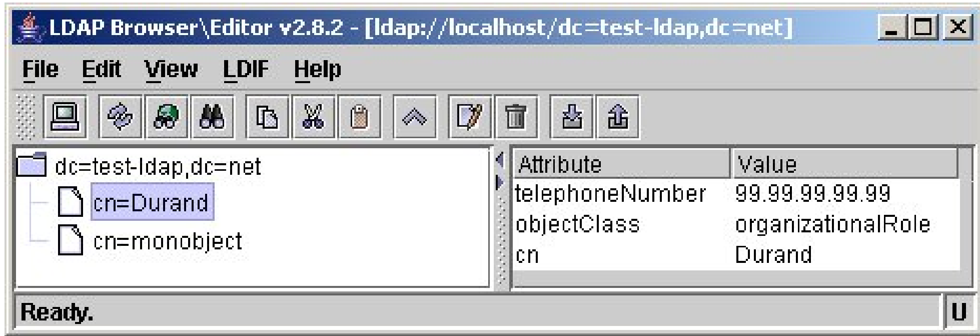
La modification d'un attribut est similaire en utilisant le type d'opération REPLACE_ATTRIBUTE

Exemple :

```
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.NamingException;
import javax.naming.directory.Attribute;
import javax.naming.directory.Attributes;
import javax.naming.directory.BasicAttribute;
```

```
import javax.naming.directory.BasicAttributes;
import javax.naming.directory.DirContext;
import javax.naming.directory.InitialDirContext;

public class TestLDAP4 {

  public static void main(String[] args) {
    Hashtable env = new Hashtable();
    env
        .put(Context.INITIAL_CONTEXT_FACTORY,
            "com.sun.jndi.ldap.LdapCtxFactory");
    env.put(Context.PROVIDER_URL, "ldap://localhost:389");
    env.put(Context.SECURITY_AUTHENTICATION, "simple");
    env.put(Context.SECURITY_PRINCIPAL, "cn=ldap-admin,dc=test-ldap,dc=net");
    env.put(Context.SECURITY_CREDENTIALS, "ldap-admin");
    DirContext dirContext;

    try {

      dirContext = new InitialDirContext(env);

      Attributes attributes = new BasicAttributes(true);
      Attribute attribut = new BasicAttribute("telephoneNumber");
      attribut.add("99.99.99.99.99");
      attributes.put(attribut);

      dirContext.modifyAttributes("cn=Durand,dc=test-ldap,dc=net",
         DirContext.REPLACE_ATTRIBUTE,attributes);
      dirContext.close();
    } catch (NamingException e) {
      System.err.println("Erreur lors de l'acces au serveur LDAP" + e);
      e.printStackTrace();
    }
    System.out.println("fin des traitements");
  }
}
```

Suite à l'exécution de ce programme, l'attribut est modifié.



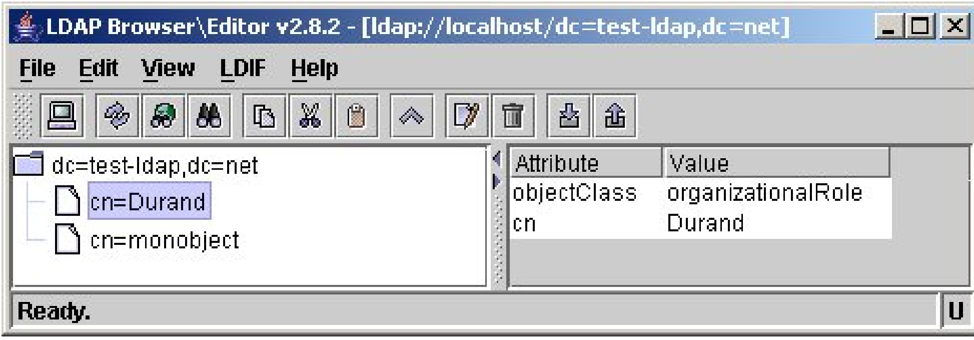
La modification d'un attribut est similaire en utilisant le type d'opération REPLACE_ATTRIBUTE

Exemple :

```
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.NamingException;
import javax.naming.directory.Attribute;
import javax.naming.directory.Attributes;
import javax.naming.directory.BasicAttribute;
import javax.naming.directory.BasicAttributes;
import javax.naming.directory.DirContext;
import javax.naming.directory.InitialDirContext;

public class TestLDAP4 {

  public static void main(String[] args) {
    Hashtable env = new Hashtable();
```

```
    env
        .put(Context.INITIAL_CONTEXT_FACTORY,
            "com.sun.jndi.ldap.LdapCtxFactory");
    env.put(Context.PROVIDER_URL, "ldap://localhost:389");
    env.put(Context.SECURITY_AUTHENTICATION, "simple");
    env.put(Context.SECURITY_PRINCIPAL, "cn=ldap-admin,dc=test-ldap,dc=net");
    env.put(Context.SECURITY_CREDENTIALS, "ldap-admin");

    DirContext dirContext;

    try {
      dirContext = new InitialDirContext(env);

      Attributes attributes = new BasicAttributes(true);
      Attribute attribut = new BasicAttribute("telephoneNumber");
      attributes.put(attribut);

      dirContext.modifyAttributes("cn=Durand,dc=test-ldap,dc=net",
         DirContext.REMOVE_ATTRIBUTE,attributes);
      dirContext.close();
    } catch (NamingException e) {
      System.err.println("Erreur lors de l'acces au serveur LDAP" + e);
      e.printStackTrace();
    }
    System.out.println("fin des traitements");
  }
}
```

Suite à l'exécution de ce programme, l'attribut est supprimé.



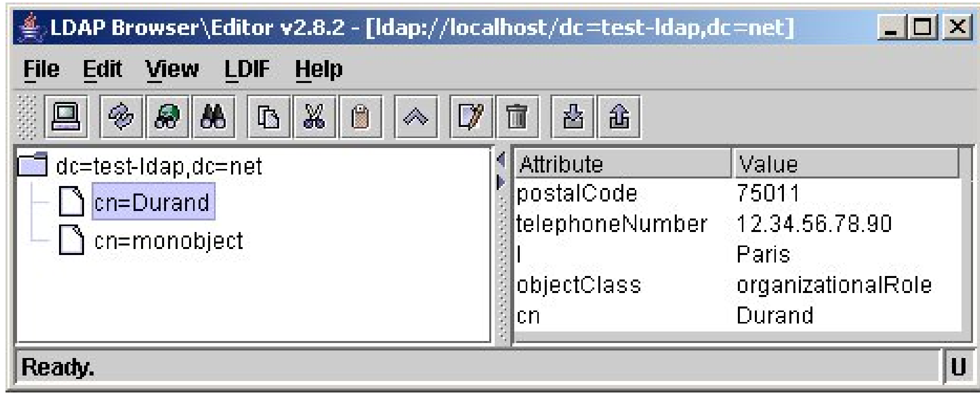
Pour réaliser plusieurs opérations, il est nécessaire d'utiliser un tableau d'objets de type ModificationItem passé en paramètre d'une version surchargée de la méthode modifyAttributes(). Dans ce cas, toutes les modifications sont effectuées ou aucune ne l'est.

Exemple :

```
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.NamingException;
import javax.naming.directory.Attribute;
import javax.naming.directory.BasicAttribute;
import javax.naming.directory.DirContext;
import javax.naming.directory.InitialDirContext;
import javax.naming.directory.ModificationItem;

public class TestLDAP5 {
  public static void main(String[] args) {
    Hashtable env = new Hashtable();
    env
        .put(Context.INITIAL_CONTEXT_FACTORY,
            "com.sun.jndi.ldap.LdapCtxFactory");
    env.put(Context.PROVIDER_URL, "ldap://localhost:389");
    env.put(Context.SECURITY_AUTHENTICATION, "simple");
    env.put(Context.SECURITY_PRINCIPAL, "cn=ldap-admin,dc=test-ldap,dc=net");
    env.put(Context.SECURITY_CREDENTIALS, "ldap-admin");

    DirContext dirContext;
```

```
    try {
      dirContext = new InitialDirContext(env);

      ModificationItem[] modifItems = new ModificationItem[3];

      Attribute mod0 = new BasicAttribute("telephonenumber","12.34.56.78.90");
      Attribute mod1 = new BasicAttribute("l", "Paris");
      Attribute mod2 = new BasicAttribute("postalCode", "75011");

      modifItems[0] = new ModificationItem(DirContext.ADD_ATTRIBUTE,mod0);
      modifItems[1] = new ModificationItem(DirContext.ADD_ATTRIBUTE,mod1);
      modifItems[2] = new ModificationItem(DirContext.ADD_ATTRIBUTE,mod2);

      dirContext.modifyAttributes("cn=Durand,dc=test-ldap,dc=net", modifItems);

      dirContext.close();
    } catch (NamingException e) {
      System.err.println("Erreur lors de l'acces au serveur LDAP" + e);
      e.printStackTrace();
    }
    System.out.println("fin des traitements");
  }
}
```

Suite à l'exécution de ce programme, les attributs sont ajoutés.



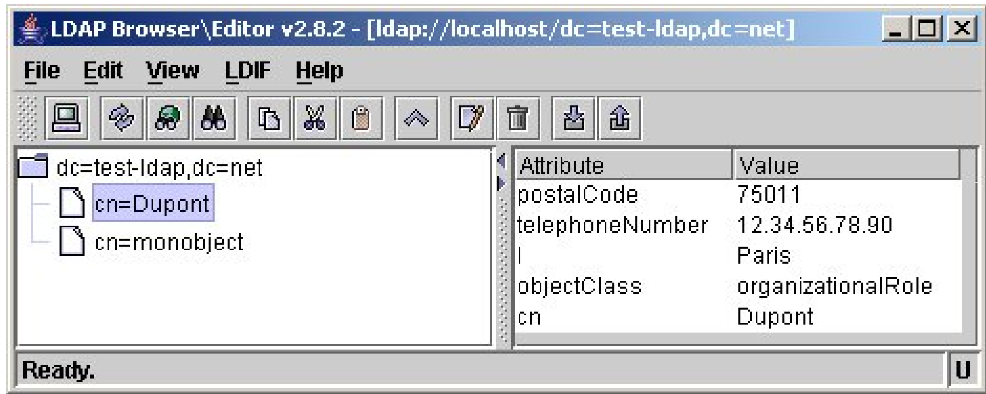
La méthode rename() permet de modifier le DN d'une entrée de l'annuaire

Exemple :

```
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.NamingException;
import javax.naming.directory.DirContext;
import javax.naming.directory.InitialDirContext;

public class TestLDAP6 {

  public static void main(String[] args) {
    Hashtable env = new Hashtable();
    env
        .put(Context.INITIAL_CONTEXT_FACTORY,
            "com.sun.jndi.ldap.LdapCtxFactory");
    env.put(Context.PROVIDER_URL, "ldap://localhost:389");
    env.put(Context.SECURITY_AUTHENTICATION, "simple");
    env.put(Context.SECURITY_PRINCIPAL, "cn=ldap-admin,dc=test-ldap,dc=net");
    env.put(Context.SECURITY_CREDENTIALS, "ldap-admin");
    DirContext dirContext;

    try {
      dirContext = new InitialDirContext(env);
      dirContext.rename("cn=Durand,dc=test-ldap,dc=net",
          "cn=Dupont,dc=test-ldap,dc=net");
      dirContext.close();
    } catch (NamingException e) {
```

```
      System.err.println("Erreur lors de l'acces au serveur LDAP" + e);
      e.printStackTrace();
    }
    System.out.println("fin des traitements");
  }
}
```

Suite à l'exécution de ce programme, le DN est modifié



Si le DN à modifier fourni en paramètre n'est pas trouvé dans l'annuaire, une exception de type javax.naming.NameNotFoundException avec le message « [LDAP: error code 32 - No Such Object] » est levée.

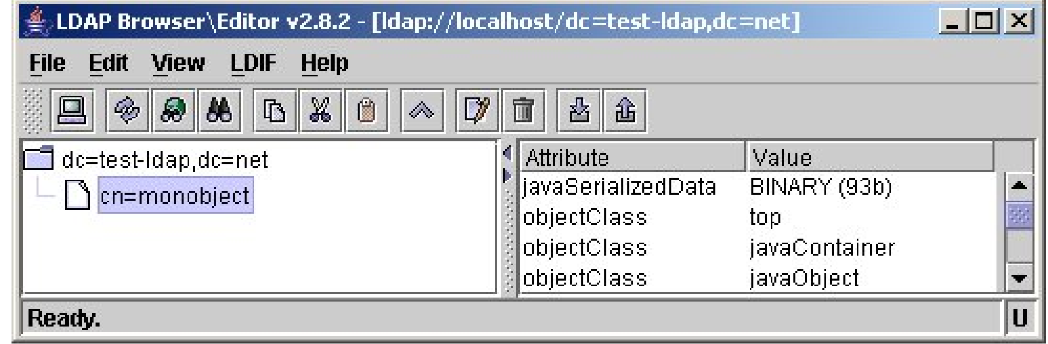
29.7.8. La suppression d'un objet

La méthode unbind() de la classe Context permet de supprimer une association entre un nom et un objet.

Exemple :

```
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.NamingException;
import javax.naming.directory.DirContext;
import javax.naming.directory.InitialDirContext;

public class TestLDAP7 {

  public static void main(String[] args) {
    Hashtable env = new Hashtable();
    env
        .put(Context.INITIAL_CONTEXT_FACTORY,
            "com.sun.jndi.ldap.LdapCtxFactory");
    env.put(Context.PROVIDER_URL, "ldap://localhost:389");
    env.put(Context.SECURITY_AUTHENTICATION, "simple");
    env.put(Context.SECURITY_PRINCIPAL, "cn=ldap-admin,dc=test-ldap,dc=net");
    env.put(Context.SECURITY_CREDENTIALS, "ldap-admin");
    DirContext dirContext;

    try {
      dirContext = new InitialDirContext(env);
      dirContext.unbind("cn=Dupont,dc=test-ldap,dc=net");
      dirContext.close();
    } catch (NamingException e) {
      System.err.println("Erreur lors de l'acces au serveur LDAP" + e);
      e.printStackTrace();
    }
    System.out.println("fin des traitements");
  }
}
```

Suite à l'exécution de ce programme, l'entrée dans l'annuaire est supprimée.



Il n'est pas possible de supprimer un objet que si ce dernier est le dernier du contexte. Une demande de suppression d'un élément qui ne soit pas le dernier, lèvera une exception de type ContextNotEmptyException avec le message « [LDAP: error code 66 - subtree delete not supported] ».

Exemple :

```
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.NamingException;
import javax.naming.directory.DirContext;
import javax.naming.directory.InitialDirContext;

public class TestLDAP8 {

  public static void main(String[] args) {
    Hashtable env = new Hashtable();
    env
        .put(Context.INITIAL_CONTEXT_FACTORY,
            "com.sun.jndi.ldap.LdapCtxFactory");
    env.put(Context.PROVIDER_URL, "ldap://localhost:389");
    env.put(Context.SECURITY_AUTHENTICATION, "simple");
    env.put(Context.SECURITY_PRINCIPAL, "cn=ldap-admin,dc=test-ldap,dc=net");
    env.put(Context.SECURITY_CREDENTIALS, "ldap-admin");
    DirContext dirContext;

    try {
      dirContext = new InitialDirContext(env);
      dirContext.destroySubcontext("dc=test-ldap,dc=net");
      dirContext.close();
    } catch (NamingException e) {
      System.err.println("Erreur lors de l'acces au serveur LDAP" + e);
      e.printStackTrace();
    }
    System.out.println("fin des traitements");
  }
}
```

29.7.9. La recherche d'associations

La méthode listBindings() permet d'obtenir une liste des associations nom/objet.

Elle renvoie un objet de type NamingEnumeration qui encapsule des objets de type Binding.

Exemple :

```
import java.util.Hashtable;
import javax.naming.Binding;
import javax.naming.Context;
import javax.naming.NamingEnumeration;
import javax.naming.NamingException;
import javax.naming.directory.DirContext;
import javax.naming.directory.InitialDirContext;
```

```java
public class TestLDAP13 {

  public static void main(String[] args) {
    Hashtable env = new Hashtable();
    env
        .put(Context.INITIAL_CONTEXT_FACTORY,
            "com.sun.jndi.ldap.LdapCtxFactory");
    env.put(Context.PROVIDER_URL, "ldap://localhost:389");
    env.put(Context.SECURITY_AUTHENTICATION, "simple");
    env.put(Context.SECURITY_PRINCIPAL, "cn=ldap-admin,dc=test-ldap,dc=net");
    env.put(Context.SECURITY_CREDENTIALS, "ldap-admin");
    DirContext dirContext;

    try {

      dirContext = new InitialDirContext(env);
      NamingEnumeration e = dirContext.listBindings("dc=test-ldap,dc=net");

      while (e.hasMore())
          {
          Binding b = (Binding) e.next();
          System.out.println("nom    : " + b.getName());
          System.out.println("objet  : " + b.getObject());
          System.out.println("classe : " + b.getObject().getClass().getName());
          }

      dirContext.close();
    } catch (NamingException e) {
      System.err.println("Erreur lors de l'acces au serveur LDAP" + e);
      e.printStackTrace();
    }
    System.out.println("fin des traitements");
  }
}
```

Exemple :

```
nom    : cn=monobject
objet  : MonObjet@1764be1
classe : MonObjet
nom    : cn=Durand
objet  : com.sun.jndi.ldap.LdapCtx@16fd0b7
classe : com.sun.jndi.ldap.LdapCtx
nom    : cn=Pierre
objet  : com.sun.jndi.ldap.LdapCtx@1ef9f1d
classe : com.sun.jndi.ldap.LdapCtx
nom    : cn=Martin
objet  : com.sun.jndi.ldap.LdapCtx@b753f8
classe : com.sun.jndi.ldap.LdapCtx
nom    : cn=Dupont
objet  : com.sun.jndi.ldap.LdapCtx@1e9cb75
classe : com.sun.jndi.ldap.LdapCtx
```

29.7.10. La recherche dans un annuaire LDAP

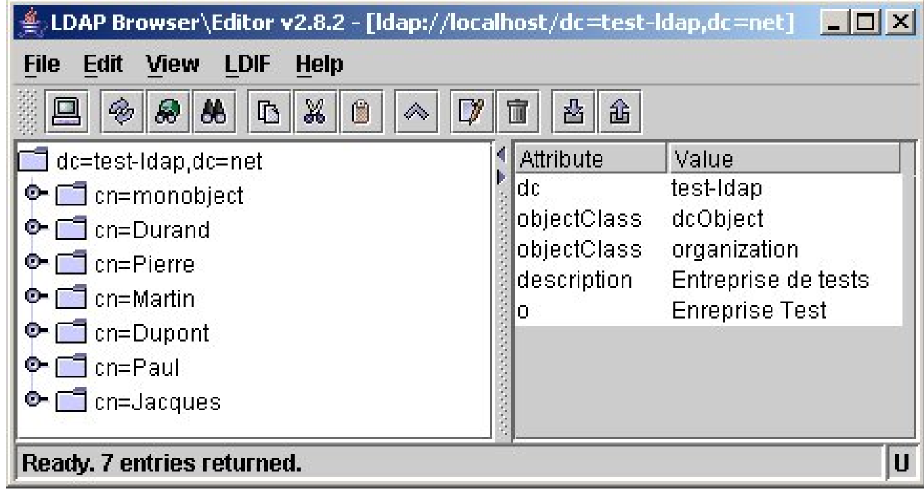
La recherche d'objets et d'informations contenues dans un objet est une des principales actions réalisées sur un annuaire.

La recherche dans un annuaire peut se faire à partir du DN d'un objet mais aussi à partir d'un ou plusieurs attributs. Cette recherche s'effectue grâce une requête de type filtre qui possède une syntaxe particulière.

La classe DirContext propose deux fonctionnalités pour effectuer des recherches :

- Une recherche à partir du DN
- Une recherche à partir d'un filtre qui permet une recherche avancée (éventuellement sur plusieurs critères)

Les exemples de cette section utilisent le jeu d'essais suivant :



La méthode getAttributes() permet d'obtenir tous les attributs d'un objet à partir de son DN.

Exemple :

```
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.NamingException;
import javax.naming.directory.Attributes;
import javax.naming.directory.DirContext;
import javax.naming.directory.InitialDirContext;

public class TestLDAP10 {
  public static void main(String[] args) {
    Hashtable env = new Hashtable();
    env
        .put(Context.INITIAL_CONTEXT_FACTORY,
            "com.sun.jndi.ldap.LdapCtxFactory");
    env.put(Context.PROVIDER_URL, "ldap://localhost:389");
    env.put(Context.SECURITY_AUTHENTICATION, "simple");
    env.put(Context.SECURITY_PRINCIPAL, "cn=ldap-admin,dc=test-ldap,dc=net");
    env.put(Context.SECURITY_CREDENTIALS, "ldap-admin");
    DirContext dirContext;

    try {
      dirContext = new InitialDirContext(env);

      Attributes attrs = dirContext.getAttributes("cn=Dupont,dc=test-ldap,dc=net");
      System.out.println("Description : " + attrs.get("description").get());

      dirContext.close();
    } catch (NamingException e) {
      System.err.println("Erreur lors de l'acces au serveur LDAP" + e);
      e.printStackTrace();
    }
    System.out.println("fin des traitements");
  }
}
```

Résultat :

```
  Description : Directeur
  fin des traitements
```

Ceci impose de connaître le DN de l'objet. JNDI propose la possibilité de rechercher un ou plusieurs objets en utilisant un filtre.

Il est possible de faire une recherche sur un ou plusieurs attributs possédant une valeur particulière par les objets retournés lors de la recherche. Cette recherche se fait en utilisant la méthode search().

Deux surcharges de la méthode search permettent la recherche à partir d'attributs :

- NamingEnumeration search(String stringName, Attributes attributesToMatch)
- NamingEnumeration search(String stringName, Attributes attributesToMatch, String [] rgstringAttributesToReturn)

Les deux méthodes permettent de retrouver un objet dont le nom est fourni en paramètre et qui possède en plus les attributs précisés.

La seconde méthode permet aussi de préciser un tableau des attributs renvoyés dans les résultats de la recherche.

Exemple :

```
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.NamingEnumeration;
import javax.naming.NamingException;
import javax.naming.directory.Attributes;
import javax.naming.directory.BasicAttribute;
import javax.naming.directory.BasicAttributes;
import javax.naming.directory.DirContext;
import javax.naming.directory.InitialDirContext;
import javax.naming.directory.SearchResult;

public class TestLDAP11 {
  public static void main(String[] args) {
    Hashtable env = new Hashtable();
    env
        .put(Context.INITIAL_CONTEXT_FACTORY,
            "com.sun.jndi.ldap.LdapCtxFactory");
    env.put(Context.PROVIDER_URL, "ldap://localhost:389");
    env.put(Context.SECURITY_AUTHENTICATION, "simple");
    env.put(Context.SECURITY_PRINCIPAL, "cn=ldap-admin,dc=test-ldap,dc=net");
    env.put(Context.SECURITY_CREDENTIALS, "ldap-admin");
    DirContext dirContext;

    try {
      dirContext = new InitialDirContext(env);

      Attributes matchattribs = new BasicAttributes(true);
      matchattribs.put(new BasicAttribute("description", "Employe"));
      NamingEnumeration resultat = dirContext.search("dc=test-ldap,dc=net", matchattribs);

      while (resultat.hasMore()) {
        SearchResult sr = (SearchResult)resultat.next();
        System.out.println("Description : " + sr.getAttributes().get("cn").get());
      }

      dirContext.close();
    } catch (NamingException e) {
      System.err.println("Erreur lors de l'acces au serveur LDAP" + e);
      e.printStackTrace();
    }
    System.out.println("fin des traitements");
  }
}
```

Résultat :

```
  Description : Pierre
  Description : Paul
  Description : Jacques
  fin des traitements
```

La recherche peut se faire à partir d'un filtre dont les spécifications sont définies dans la RFC 2254.

Le filtre est une expression logique qui précise les critères de recherche. La syntaxe de ce filtre est composée de conditions utilisées avec des opérateurs logiques. Un opérateur doit être précisé avant la ou les conditions sur lesquelles il agit. La syntaxe est donc de la forme :

```
(operateur(condition)(condition)...))
```

<table>
<tr><th>Opérateur</th><th>Condition</th><th>Exemple</th><th>Description</th></tr>
<tr><td>=</td><td>Egalité</td><td><code>(sn=test)</code></td><td>tous les objets dont l'attribut sn vaut test</td></tr>
<tr><td>></td><td>Plus grand que</td><td><code>(sn>test)</code></td><td>tous les objets sont l'attribut sn est alphabétiquement plus grand que test</td></tr>
<tr><td>>=</td><td>Plus grand ou égal à</td><td><code>(sn>=test)</code></td><td>tous les objets sont l'attribut sn est alphabétiquement plus grand ou égal à test</td></tr>
<tr><td><</td><td>Plus petit que</td><td><code>(sn<test)</code></td><td>tous les objets sont l'attribut sn est alphabétiquement plus petit que test</td></tr>
<tr><td><=</td><td>Plus petit ou égal à</td><td><code>(sn<=test)</code></td><td>tous les objets sont l'attribut sn est alphabétiquement plus petit ou égal à test</td></tr>
<tr><td>=*</td><td>Est présent</td><td><code>(sn=*)</code></td><td>tous les objets possédant un attribut sn</td></tr>
<tr><td>*</td><td>Aucun ou plusieurs caractères quelconques</td><td><code>(sn=test*),</code>
<code>(sn=*test*),</code>
<code>(sn=*test)</code></td><td>respectivement tous les objets dont l'attribut sn contient commence par test, contient test ou termine par test</td></tr>
<tr><td>&</td><td>ET</td><td><code>(&(sn=test) (cn=test))</code></td><td>tous les objets dont l'attribut sn et cn valent test</td></tr>
<tr><td>|</td><td>OU</td><td><code>(|(sn=test) (cn=test))</code></td><td>tous les objets dont l'attribut sn ou cn valent test</td></tr>
<tr><td>!</td><td>NON</td><td><code>(!(sn=test))</code></td><td>tous les objets sont l'attribut sn est différent de test</td></tr>
</table>

Quatre autres surcharges de la méthode search() permettent de faire une recherche à partir d'un filtre.

- NamingEnumeration search(Name name, String filterExpr, Object[] filterArgs, SearchControls cons)
- NamingEnumeration search(String name, String filterExpr, Object[] filterArgs, SearchControls cons)
- NamingEnumeration search(Name name, String filter, SearchControls cons)
- NamingEnumeration search(String name, String filter, SearchControls cons)

La classe SearchControls encapsule des informations de contrôle sur la recherche à effectuer notamment :

- searchScope : la portée de la recherche (OBJECT_SCOPE, ONELEVEL_SCOPE, SUBTREE_SCOPE)
- countLimit : le nombre maximum d'occurrences renvoyées par la recherche
- timeLimit : durée maximale en milliseconde de la recherche
- returningAttributes : tableau des attributs retourné par la recherche
- returningObjFlag : précise si les objets correspondant à la recherche sont retournés dans les résultats

Le résultat de la recherche est encapsulé dans un objet de type NamingEnumeration : cet objet est une énumération d'objet de type SearchResult.

Exemple :

```
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.NamingEnumeration;
import javax.naming.NamingException;
import javax.naming.directory.DirContext;
import javax.naming.directory.InitialDirContext;
import javax.naming.directory.SearchControls;
import javax.naming.directory.SearchResult;
public class TestLDAP12 {
  public static void main(String[] args) {
    Hashtable env = new Hashtable();
    env
        .put(Context.INITIAL_CONTEXT_FACTORY,
            "com.sun.jndi.ldap.LdapCtxFactory");
    env.put(Context.PROVIDER_URL, "ldap://localhost:389");
    env.put(Context.SECURITY_AUTHENTICATION, "simple");
    env.put(Context.SECURITY_PRINCIPAL, "cn=ldap-admin,dc=test-ldap,dc=net");
    env.put(Context.SECURITY_CREDENTIALS, "ldap-admin");
    DirContext dirContext;
```

```
    try {
      dirContext = new InitialDirContext(env);
      SearchControls searchControls = new SearchControls();
      searchControls.setSearchScope(SearchControls.SUBTREE_SCOPE);
      NamingEnumeration resultat = dirContext.search("dc=test-ldap,dc=net",
            "(cn=Martin)", searchControls);
      while (resultat.hasMore()) {
        SearchResult sr = (SearchResult)resultat.next();
        System.out.println("Description : " + sr.getAttributes().get("cn").get()
           +", "+sr.getAttributes().get("description").get());
      }
      dirContext.close();
    } catch (NamingException e) {
      System.err.println("Erreur lors de l'acces au serveur LDAP" + e);
      e.printStackTrace();
    }
    System.out.println("fin des traitements");
  }
}
```

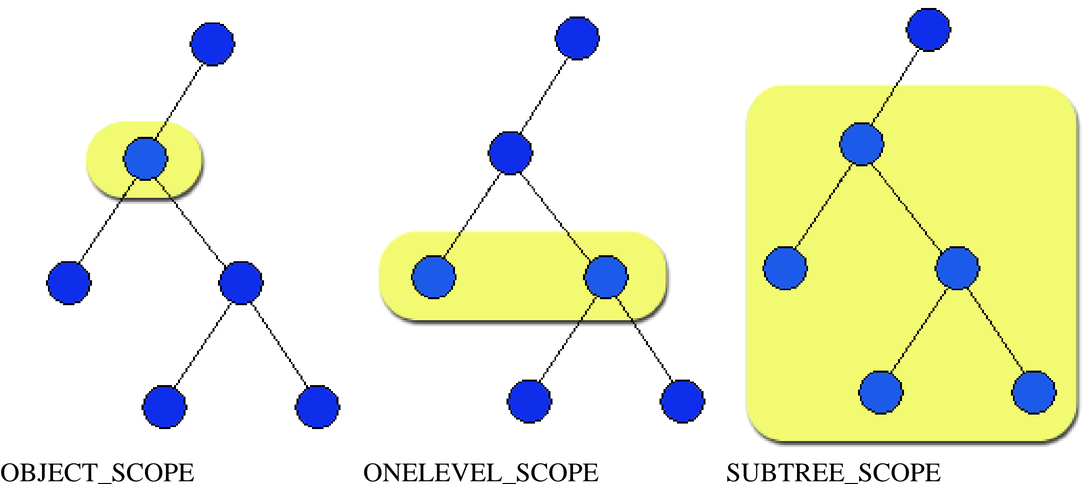
Résultat :

```
Description : Martin, Chef d'equipe
fin des traitements
```

Lors d'une recherché, faut préciser le noeud de départ (base object) de la recherche et la portée de cette recherche (scope). La portée permet de définir quels seront les noeuds concernés par la recherche.

Trois portées de recherche sont définies :

Portée	Définition
OBJECT_SCOPE	Cette portée ne concerne que le noeud de départ lui-même. Cette portée est utile pour recherche des attributs sur un objet
ONELEVEL_SCOPE	Cette portée concerne tous les noeuds d'un même niveau
SUBTREE_SCOPE	C'est la portée la plus grande puisqu'elle inclut le noeud de départ et tous ces noeuds fils



Exemple :

```
    SearchControls ctls = new SearchControls();
    ctls.setSearchScope(SearchControls.SUBTREE_SCOPE);
```

Par défaut, tous les attributs des objets trouvés sont retournés. Il est possible de limiter les attributs retournés en créant un tableau des clés des attributs concernés. Il suffit alors de passer se paramètre à la méthode setReturningAttributes() de l'instance de la classe SearchControls.

Exemple :

```
String[] attributIDs = {"cn", "description"};
searchControls.setReturningAttributes(attributIDs);
```

Il est possible de limiter le nombre d'objets retournés dans le résultat de la recherche. Il suffit de fournir en paramètre de la méthode setCountLimit() de l'instance de la classe SearchControls le nombre d'objets maximum retournés.

Exemple :

```
searchControls.setCountLimit(1);
```

Attention : une exception de type javax.naming.SizeLimitExceededException avec le message « [LDAP: error code 4 - Sizelimit Exceeded] » est levée si la limite est dépassée par le nombre d'objets trouvés.

Pour obtenir dans l'objet de NamingEnumeration

29.8. JNDI et J2EE

J2EE utilise énormément JNDI de façon implicite ou explicite notamment pour proposer des références vers des ressources nécessaires aux applications.

Chaque conteneur J2EE utilise en interne un service accessible via JNDI pour stocker des informations sur les applications et les composants. Généralement l'utilisation de JNDI dans une application J2EE se fait en utilisant ce service du conteneur.

Ces informations sont essentiellement des données de configuration : interface Home des EJB, DataSource pour accès à des bases de données, ... Ceci permet de rendre dynamique la recherche de composants de l'application.

Plusieurs technologies mises en oeuvre dans J2EE font un usage de JNDI : par exemple JDBC, EJB, JMS, ...

Par exemple, JDBC utilise JNDI pour stocker des objets de type DataSource qui encapsulent les informations utilisent à la connexion à la source de données. Cette utilisation a été proposée à partir du package optionnel JDBC 2.0. Son utilisation n'est pas obligatoire mais elle est fortement recommandée.

Comme JDBC, JMS recommande de stocker les informations concernant les files (queues) et les sujets (topics) dans un annuaire et de les rechercher via JNDI.

Les EJB stockent aussi leur référence vers leur interface home dans l'annuaire du serveur d'application pour permette à un client d'obtenir une référence sur l'EJB.

J2EE propose dans ces spécifications des règles de nommage pour certains objets ou composants J2EE dans l'annuaire pour permettre de standardiser les pratiques.

30. Scripting

30.1. L'API Scripting

Java SE 6.0 intègre la possibilité d'utiliser des moteurs de scripting suite à l'intégration des spécifications de la JSR 223.

La JSR 223 a pour but d'intégrer des possibilités de scripting dans les applications Java en permettant :

- L'intégration de moteurs de scripting
- La possibilité pour ces moteurs d'accéder à la plate-forme Java
- L'ajout d'une console permettant l'exécution de script en mode ligne de commande (jrunscript)

Les classes et interfaces de cette fonctionnalité sont regroupées dans le package javax.script.

L'API propose un support pour tous les moteurs de scripting compatible avec elle.

Java SE 6.0 intègre en standard le moteur de scripting Rhino version 1.6 R2 qui propose un support pour le langage Javascript.

La gestion des moteurs utilisables se fait via la classe ScriptEngineManager : elle permet d'obtenir la liste des objets de type ScriptEngineFactory de chaque moteur de scripting installé. Ces méthodes ne sont pas statiques, il est donc nécessaire d'instancier un objet de type ScriptEngineManager pour les utiliser.

30.1.1. La mise en oeuvre de l'API

Des fabriques permettent l'instanciation d'un objet de type ScriptEngine qui encapsule le moteur de scripting.

Exemple :

```
package com.jmd.tests.java6;

import java.util.List;
import javax.script.ScriptEngineFactory;
import javax.script.ScriptEngineManager;

public class ListerScriptEngine {

  public static void main(String args[]) {
    ScriptEngineManager manager = new ScriptEngineManager();
    List<ScriptEngineFactory> factories = manager.getEngineFactories();

    for (ScriptEngineFactory factory : factories) {
      System.out.println("Name : " + factory.getEngineName());
      System.out.println("Version : " + factory.getEngineVersion());
      System.out.println("Language name : " + factory.getLanguageName());
      System.out.println("Language version : " + factory.getLanguageVersion());
      System.out.println("Extensions : " + factory.getExtensions());
      System.out.println("Mime types : " + factory.getMimeTypes());
      System.out.println("Names : " + factory.getNames());
    }
  }
```

```
}
```

Résultat :

```
Name : Mozilla Rhino
Version : 1.6 release 2
Language name : ECMAScript
Language version : 1.6
Extensions : [js]
Mime types:[application/javascript, application/ecmascript, text/javascript, text/ecmascript]
Names : [js, rhino, JavaScript, javascript, ECMAScript, ecmascript]
```

Les propriétés Extensions, MimeType et Names sont importantes car elles sont utilisées pour obtenir une instance de la classe ScriptEngine.

Le ScriptEngineManager permet d'obtenir directement une instance du moteur de scripting à partir d'un nom, d'une extension et d'un type mime particulier respectivement grâce aux méthodes getEngineByName(), getEngineByExtension(), et getEngineByMimeType().

Exemple :

```
package com.jmd.tests.java6;

import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;

public class TestScriptEngine {
  public static void main(String args[]) {
    ScriptEngineManager manager = new ScriptEngineManager();
    ScriptEngine moteur1 = manager.getEngineByName("rhino");
    ScriptEngine moteur2 = manager.getEngineByExtension("js");
    ScriptEngine moteur3 = manager.getEngineByName("test");
    if (moteur3== null) {
      System.out.println("Impossible de trouver le moteur test ");
    }
  }
}
```

Si aucune fabrique ne correspond au paramètre fourni alors l'instance de type ScriptEngine retournée est null.

30.1.2. Ajouter d'autres moteurs de scripting

Il est possible d'ajouter d'autres moteurs de scripting. Le projet scripting hébergé par java.net propose l'encapsulation de nombreux moteurs de scripting pour l'utilisation avec l'API Scripting. https://scripting.dev.java.net/

Il faut télécharger le fichier jsr223-engines.zip. Cette archive contient un répertoire pour chaque moteur. Il faut ajouter le fichier build/xxx-engine.jar au classpath ou xxx est le nom du moteur.

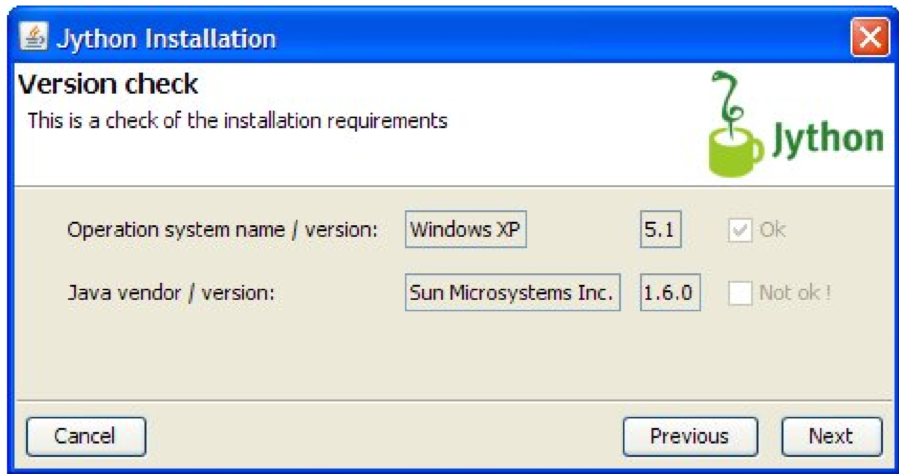
Résultat de l'exécution :

```
Name : Mozilla Rhino
Version : 1.6 release 2
Language name : ECMAScript
Language version : 1.6
Extensions : [js]
Mime types:[application/javascript, application/ecmascript, text/javascript, text/ecmascript]
Names : [js, rhino, JavaScript, javascript, ECMAScript, ecmascript]
Name : jython
Version : 2.1
Language name : python
Language version : 2.1
Extensions : [jy, py]
Mime types : []
```

```
Names : [jython, python]
```

Il est nécessaire pour instancier le moteur que celui-ci soit présent dans le classpath. Dans le cas de jython, il faut ajouter le fichier jython.jar dans le classpath (). Pour cela, il faut télécharger le fichier jython_Release_2_2alpha1.jar et l'exécuter en double cliquant dessus. Le programme d'installation utilise un assistant :

- Sur la page « Welcome to Jython », cliquez sur le bouton « Next »
- Sur la page « Installation type », laissez All sélectionné et cliquez sur Next



- Sur la page « Version check », cliquez sur Next
- Sur la page « License agreement », lisez la licence et si vous l'acceptez cliquez sur « I accept » et sur le bouton « Next »
- Sur la page « Target directory » modifiez le répertoire d'installation au besoin et cliquez sur Next. Si le répertoire n'existe pas, cliquez sur OK puis de nouveau sur le bouton Next
- Sur la page « Overview (summary of options) », cliquez sur Next pour démarrer l'installation
- Sur la page « Read me », cliquez sur Next
- Cliquez sur Finish pour terminer l'installation.

Ajoutez le fichier jython.jar contenu dans le répertoire d'installation au classpath de l'application. Il est alors possible de créer une instance du moteur de script Jython.

Exemple :

```
package com.jmd.tests.java6;

import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;

public class TestJython {
  public static void main(String args[]) {
    try {
      ScriptEngineManager manager = new ScriptEngineManager();
      ScriptEngine moteur = manager.getEngineByName("jython");
      if (moteur== null) {
        System.out.println("Impossible de trouver le moteur jython ");
```

```
      }
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Si le fichier jython.jar n'est pas présent dans le classpath une exception est levée.

Exemple :

```
Exception in thread "main" java.lang.NoClassDefFoundError: org/python/core/PyObject
at com.sun.script.jython.JythonScriptEngineFactory.getScriptEngine(
JythonScriptEngineFactory.java:132)
at javax.script.ScriptEngineManager.getEngineByName(ScriptEngineManager.java:225)
at com.jmd.tests.java6.TestJython.main(TestJython.java:11)
```

30.1.3. L'évaluation d'un script

La classe ScriptEngine propose plusieurs surcharges de la méthode eval() pour exécuter un script. Ces surcharges attendent en paramètre le script sous la forme d'une chaîne de caractères ou d'un flux de type Reader.

Exemple :

```
package com.jmd.tests.java6;

import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;

public class TestJython {
  public static void main(String args[]) {
    try {
      ScriptEngineManager manager = new ScriptEngineManager();
      ScriptEngine moteur = manager.getEngineByName("jython");
      if (moteur == null) {
        System.out.println("Impossible de trouver le moteur jython ");
      } else {
        moteur.eval("print \"test\"");
      }
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

La méthode eval() peut lever une exception de type javax.script.ScriptException si une erreur est détectée par le moteur dans le script

Exemple :

```
package com.jmd.tests.java6;

import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import javax.script.ScriptException;

public class TestRhino {
  public static void main(String args[]) {
    ScriptEngineManager manager = new ScriptEngineManager();
    ScriptEngine moteur = manager.getEngineByName("rhino");
    try {
      moteur.eval("alert('test');");
    } catch (ScriptException e) {
```

```
            e.printStackTrace();
        }
    }
}
```

Résultat :

```
javax.script.ScriptException: sun.org.mozilla.javascript.internal.EcmaError: ReferenceError:
"Alert" n'est pas défini (<Unknown source>#1) in <Unknown source> at line number 1
at com.sun.script.javascript.RhinoScriptEngine.eval(RhinoScriptEngine.java:110)
at com.sun.script.javascript.RhinoScriptEngine.eval(RhinoScriptEngine.java:124)
at javax.script.AbstractScriptEngine.eval(AbstractScriptEngine.java:247)
at com.jmd.tests.java6.TestRhino.main(TestRhino.java:13)
```

Deux surcharges de la méthode eval() attendant en paramètre un objet de type Bindings. C'est un objet de type Map qui permet de passer des objets Java au script.

Exemple :

```
package com.jmd.tests.java6;

import javax.script.Bindings;
import javax.script.ScriptContext;
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import javax.script.ScriptException;

public class TestBindings {

  public static void main(String args[]) {
    ScriptEngineManager manager = new ScriptEngineManager();
    ScriptEngine moteur = manager.getEngineByName("rhino");
    try {
      Bindings bindings = moteur.getBindings(ScriptContext.ENGINE_SCOPE);
      bindings.clear();
      bindings.put("entree","valeur");
      moteur.eval("var sortie = '';"+
        " sortie = entree + ' modifiee '", bindings);
      String resultat = (String)bindings.get("sortie");
      System.out.println("resultat = "+resultat);
    } catch (ScriptException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
resultat = valeur modifiee
```

La classe ScriptEngine possède deux méthodes pour faciliter l'utilisation des Bindings : les méthodes put() et get() pour respectivement passer un objet au script et obtenir un objet du script.

Exemple :

```
package com.jmd.tests.java6;

import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import javax.script.ScriptException;

public class TestBindings2 {
  public static void main(String args[]) {
    ScriptEngineManager manager = new ScriptEngineManager();
    ScriptEngine moteur = manager.getEngineByName("rhino");
    try {
      moteur.put("entree","valeur");
```

```
      moteur.eval("var sortie = '';"+
        " sortie = entree + ' modifiee '");
      String resultat = (String)moteur.get("sortie");
      System.out.println("resultat = "+resultat);
    } catch (ScriptException e) {
      e.printStackTrace();
    }
  }
}
```

Deux surcharges de la méthode eval() attendent en paramètre un objet de type ScriptContext qui permet de préciser la porté des Bindings.

Il existe deux portées prédéfinies :

- ScriptContext.GLOBAL_SCOPE : portée pour tous les moteurs
- ScriptContext.ENGINE_SCOPE : portée pour le moteur courant uniquement

Il est possible de préciser le contexte par défaut du moteur en utilisant la méthode SetContext() de la classe ScriptEngine.

La méthode getBindings() permet d'obtenir les bindings pour la portée fournie en paramètre.

30.1.4. L'interface Compilable

Les scripts sont généralement interprétés : ils doivent donc être lus, validés et évalués avant d'être exécutés. Ces opérations peuvent être coûteuses en ressources et en temps.

L'interface Compilable propose de compiler ces scripts afin de rendre leur prochaine exécution plus rapide. L'implémentation de cette interface par un moteur de scripting est optionnelle : il faut donc vérifier que l'instance du moteur de scripting implémente cette interface et le caster vers le type Compilable avant d'utiliser ces fonctionnalités.

La méthode compile() réalise une compilation du script et retourne un objet de type CompiledScript en cas de succès.

Le script compilé est exécuté avec la méthode eval() de la classe CompiledScript.

Exemple :

```
package com.jmd.tests.java6;

import javax.script.Bindings;
import javax.script.Compilable;
import javax.script.CompiledScript;
import javax.script.ScriptContext;
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import javax.script.ScriptException;

public class TestCompilable {

  public static void main(String args[]) {
    ScriptEngineManager manager = new ScriptEngineManager();
    ScriptEngine moteur = manager.getEngineByName("rhino");
    try {
      Bindings bindings = moteur.getBindings(ScriptContext.ENGINE_SCOPE);
      bindings.clear();
      bindings.put("compteur", 1);

      if (moteur instanceof Compilable) {
        Compilable moteurCompilable = (Compilable) moteur;
        CompiledScript scriptCompile = moteurCompilable
          .compile("var sortie = '';"
          + "sortie = 'chaine' + compteur;"
          + "compteur++;");
```

```
        for (int i = 1; i < 11; i++) {
          scriptCompile.eval(bindings);
          String resultat = (String) bindings.get("sortie");
          System.out.println("valeur " + i + " = " + resultat);
        }
      } else {
      System.err
        .println("Le moteur n'implemente pas l'interface Compilable");
      }
    } catch (ScriptException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
valeur 1 = chaine1
valeur 2 = chaine2
valeur 3 = chaine3
valeur 4 = chaine4
valeur 5 = chaine5
valeur 6 = chaine6
valeur 7 = chaine7
valeur 8 = chaine8
valeur 9 = chaine9
valeur 10 = chaine10
```

L'utilisation de cette fonctionnalité est particulièrement intéressante pour des exécutions répétées du script.

30.1.5. L'interface Invocable

Cette interface permet d'invoquer une fonction définie dans le code source du script.

Dès qu'une fonction a été évaluée par le moteur de scripting, elle peut être invoquée grâce à la méthode invoke() de l'interface Invocable. L'implémentation de cette interface par un moteur de scripting est optionnelle : il faut donc vérifier avant d'utiliser ces fonctionnalités si le moteur implémente cette interface.

Il est possible de fournir des paramètres à la fonction invoquée.

Exemple :

```
package com.jmd.tests.java6;

import javax.script.Bindings;
import javax.script.Invocable;
import javax.script.ScriptContext;
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import javax.script.ScriptException;

public class TestInvocable {

  public static void main(String args[]) {
    ScriptEngineManager manager = new ScriptEngineManager();
    ScriptEngine moteur = manager.getEngineByName("rhino");
    try {

      moteur.eval("function afficher(valeur) {"
        + "var sortie = '';"
        + "sortie = 'chaine' + valeur;"
        + "return sortie;"
        + "}");

      if (moteur instanceof Invocable) {
        Invocable moteurInvocable = (Invocable) moteur;
```

```
        Object resultat = moteurInvocable.invokeFunction("afficher",
          new Integer(10));
        System.out.println("resultat = " + resultat);
      } else {
        System.err.println("Le moteur n'implemente pas l'interface Invocable");
      }

    } catch (ScriptException e) {
      e.printStackTrace();
    } catch (NoSuchMethodException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
resultat = chaine10
```

La méthode getInterface() de l'interface Invocable permet d'obtenir dynamiquement un objet dont la ou les méthodes sont codées dans le script.

L'exemple ci dessous va définir une fonction run() qui sera invoquée dans un thread en utilisant la méthode getInterface() avec en paramètre un objet de type Class qui encapsule la classe Runnable

Exemple :

```
package com.jmd.tests.java6;

import javax.script.Invocable;
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import javax.script.ScriptException;

public class TestInvocable2 {

  public static void main(String args[]) {
    ScriptEngineManager manager = new ScriptEngineManager();
    ScriptEngine moteur = manager.getEngineByName("rhino");
    try {

      moteur.eval("function run(){"
        + "for (i = 0 ; i < 1000 ; i ++) {"
        + "print('run'+i);"
        + "}"
        + "}");

      if (moteur instanceof Invocable) {
        Invocable moteurInvocable = (Invocable) moteur;
        Runnable runnable = moteurInvocable.getInterface(Runnable.class);
        Thread thread = new Thread(runnable);
        thread.start();
        for (int i = 0; i < 1000; i++) {
          System.out.println("main" + i);
        }
        thread.join();
      } else {
        System.err.println("Le moteur n'implemente pas l'interface Invocable");
      }

    } catch (ScriptException e) {
      e.printStackTrace();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```

Extrait du résultat :

```
main988
run174run175main989
main990
main991
main992
main993
main994
main995
main996
main997
main998
main999
run176run177run178run179run180
```

L'itération du programme s'exécute beaucoup plus rapidement que le thread : l'appel à la méthode join() du thread permet d'attendre la fin de son exécution avant de terminer l'application.

30.1.6. La commande jrunscript

La commande jrunscript est un outil du JDK utilisable en ligne de commande qui permet d'exécuter des scripts.

Remarque : pour pouvoir utiliser cette commande, il est nécessaire d'ajouter dans le path le chemin du répertoire bin du JDK.

Les options -help et -? permettent d'obtenir la liste des options de la commande.
L'option -q permet de connaître la liste des moteurs de scripting utilisables.
Les options -cp et -classpath permettent de préciser le classpath qui sera utilisé.

Si seul le moteur de scripting par défaut est installé alors il n'est pas utile de préciser le moteur à utiliser. Dans le cas contraire, il est nécessaire de préciser le moteur à utiliser grâce à l'option -l

Exemple :

```
C:\>jrunscript -q
Language ECMAScript 1.6 implemention "Mozilla Rhino" 1.6 release 2
```

Sans autre argument, la commande jrunscript affiche un prompt qui permet de saisir le script à évaluer.

Exemple :

```
C:\>jrunscript
js> var i = 10* 10;
js> i;
100.0
js> i
100.0
js> i++;
100.0
js> i
101.0
js> print i;
script error: sun.org.mozilla.javascript.internal.EvaluatorException: il manque
';' avant une instruction (<STDIN>#1) in <STDIN> at line number 1
js>
```

31. JMX (Java Management Extensions)

JMX est l'acronyme de Java Management Extensions. Historiquement, cette API se nommait JMAPI (Java Management API). La version 5.0 de Java a ajouté l'API JMX 1.2 dans la bibliothèque de classes standards.

JMX est une spécification qui définit une architecture, une API et des services pour permettre de surveiller et de gérer des ressources en Java. JMX permet de mettre en place, en utilisant un standard, un système de surveillance et de gestion d'une application, d'un service ou d'une ressource sans avoir à fournir beaucoup d'effort.

JMX permet de construire et de mettre en oeuvre une solution de gestion de ressources sous une forme modulaire grâce à des composants. Son but est de proposer un standard pour faciliter le développement de systèmes de contrôle, d'administration et de supervision des applications et des ressources.

JMX peut permettre de configurer, gérer et de maintenir une application durant son exécution en fonction des fonctionnalités développées. Il peut aussi favoriser l'anticipation de certains problèmes par une information sur les événements critiques de l'application ou du système.

Java SE version 5.0 intègre JMX 1.2 et JMX Remote API 1.0. Il existe aussi une implémentation téléchargeable indépendamment pour J2SE 1.4

La plupart des principaux serveurs d'applications Java EE utilisent JMX pour la surveillance et la gestion de leurs composants.

31.1. La présentation de JMX

JMX est une spécification : pour la mettre oeuvre, il faut obligatoirement utiliser une implémentation : Sun propose une implémentation de référence, mais il est aussi possible d'utiliser JBossMX ou MX4J par exemple.

JMX est une API standard qui permet de gérer, de contrôler et de surveiller des applications, des composants, des services et même la JVM ou des périphériques dans la plate-forme Java.

Ainsi les utilisations possibles de JMX sont nombreuses, par exemple :

- consulter et modifier les paramètres de la configuration
- calculer et diffuser des statistiques d'utilisation
- émettre des événements lors de changements d'état ou d'erreur
- ...

JMX est architecturé en couches ce qui permet de séparer les responsabilités des différents composants utilisés. Ceci permet aussi d'assurer une meilleure extensibilité.

L'architecture de JMX est composée de trois niveaux :

- Instrumentation : les ressources sont instrumentées grâce à des objets de type MBean
- Agent : les MBeans enregistrés sont gérés par le MBeanServer d'un agent JMX
- Distributed services : une IHM, par exemple sous la forme d'une application tierce de gestion des ressources, interagit avec les MBeans grâce à l'agent JMX

L'architecture de JMX permet de faire de l'administration en locale ou à distance. JMX permet un accès distant pour permettre à une application de gestion d'interagir avec l'application instrumentée via l'agent et le MBeans.

JMX est spécifiée dans plusieurs JSR :

- JSR 003 : Java Management Extensions Instrumentation and Agent Specification
- JSR 160 : Java Management Extensions Remote API
- JSR 174 : Monitoring and management Specification for the JVM
- JSR 255 : version 2.0 de JMX
- JSR 262 : Web services connector for JMX agents
- JSR 146 : WBEM services : JMX provider protocol adapter
- JSR 070 : IIOP protocol adaptor for JMX

La version 1.1 des spécifications de JMX ne détaille que la partie Instrumentation et Agent.

L'instrumentation via JMX peut être ajoutée dans tout programme Java simplement en ajoutant du code aux classes existantes. Cette instrumentation concerne des propriétés, des opérations et des événements qui peuvent être exposés aux agents.

JMX est largement utilisé notamment dans les serveurs d'applications par exemple.

La plupart de applications et notamment celles exécutées côte serveur ont besoins d'être administrées pour :

- obtenir des informations sur l'activité de l'application (audit)
- modifier des paramètres de configuration sans redémarrer l'application (configuration)
- anticiper de futurs problèmes selon la valeur de certains indicateurs (surveillance/monitoring)

L'avantage de JMX est de normaliser l'API de management et de proposer une exploitation de cette API via des agents pour utiliser différents protocoles pour dialoguer avec les serveurs de MBeans.

Depuis l'intégration de JMX dans la version 1.5 de Java, l'utilisation de JMX par n'importe quelle application est possible en standard. Les fonctionnalités de base de JMX étant faciles à mettre oeuvre, il devient aisé de surveiller et administrer une application depuis un client JMX distant.

JMX est un framework pour la surveillance et l'administration de ressources ou d'applications à usage général : il ne propose par exemple aucune structure de données spécifiques au monitoring ou à la gestion.

L'API JMX est regroupée dans le package javax.management et ses sous packages :

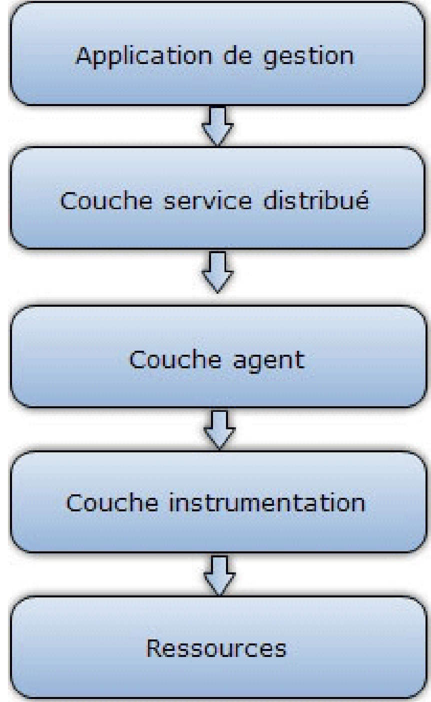
JSR	Packages
JSR 003	javax.management javax.management.loading javax.management.modelmbean javax.management.monitor javax.management.openmbean javax.management.relation javax.management.timer
JSR 160	javax.management.remote javax.management.remote.rmi
JSR 174	java.lang.management

La page web officielle relative à JMX est à l'url http://java.sun.com/products/JavaManagement/

31.2. L'architecture de JMX

L'architecture de JMX se compose de plusieurs niveaux :

- Services distribués : cette couche définit la partie IHM. C'est généralement une application qui permet de consulter les données relatives à l'application et interagir avec elles. Cette couche utilise des connecteurs et des adaptateurs de protocoles pour permettre à des outils de gestion de se connecter à un agent.
- Agent : cette couche définit un serveur de MBeans qui gère les MBeans, propose des fonctionnalités sous la forme d'un agent JMX et assure la communication avec la couche services distribués grâce à des Connectors et des Adapters
- Instrumentation : cette couche définit des MBeans qui permettent l'instrumentation d'une ressource (application, service, composant, objet, appareil, ...) grâce à des attributs, des opérations et des événements. La ressource peut être écrite en Java ou la ressource peut être encapsulée par une classe qui va communiquer avec la ressource (wrapper). Une ressource peut être instrumentée par un ou plusieurs MBeans. Un dynamique MBean implémente une interface particulièrement qui permet plus de flexibilité à l'exécution. Les MBeans n'ont pas besoin de référence sur l'agent qui va les gérer.
- Ressources gérées (composants de l'application, services, périphériques, ...) : cette couche n'est pas directement concernée par l'API JMX

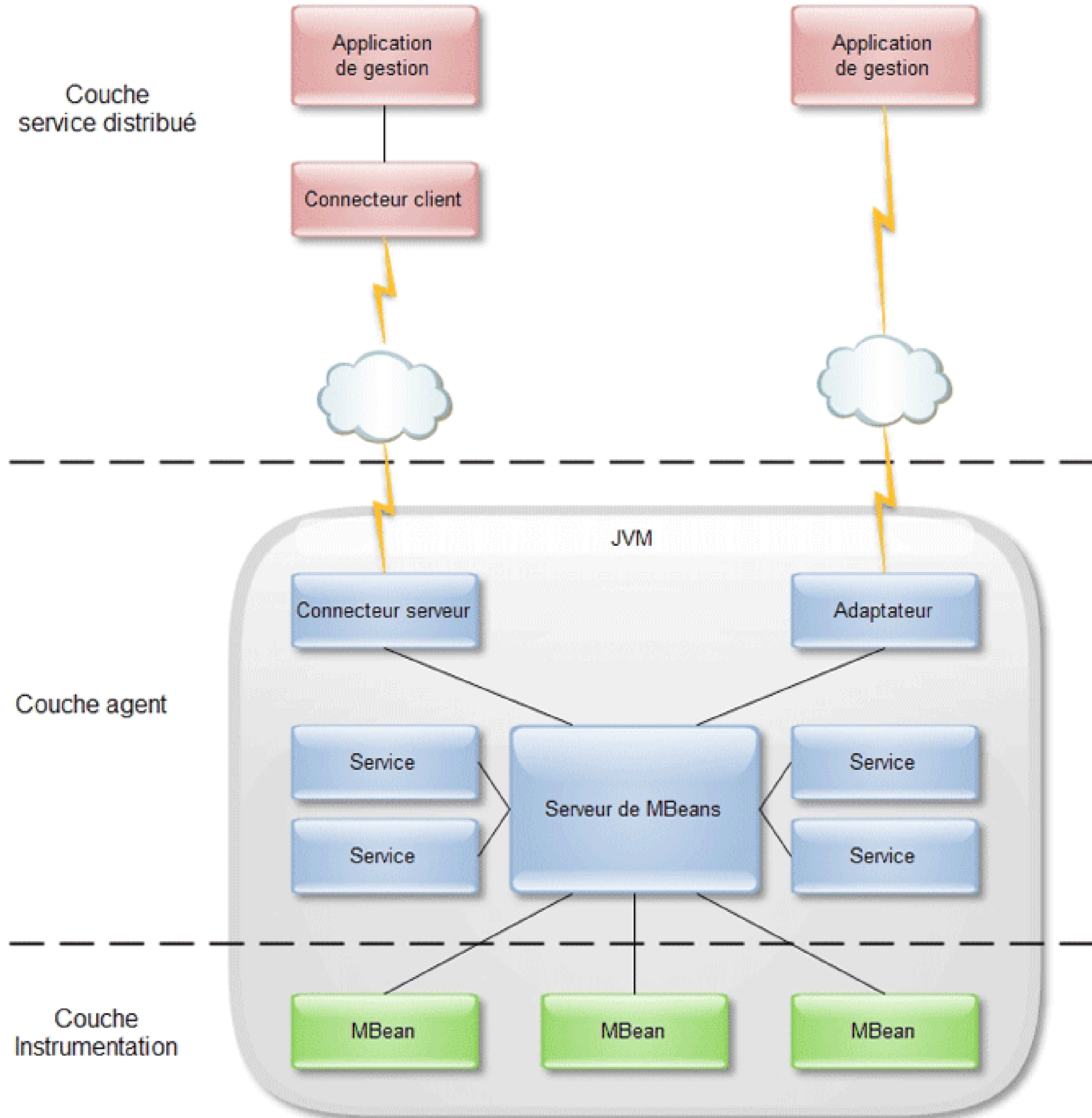


L'architecture de l'API JMX est composée des trois premières parties. Les couches instrumentation et agent sont spécifiées par la JSR 003.

La couche Remote management ou distributed services est spécifiée par la JSR 160.

La technologie JMX propose une architecture en trois couches pour permettre à des applications de gestion de gérer des ressources :

- instrumentation : des ressources (applications, services, composants, appareils, ...) sont instrumentée avec des objets Java de type MBean. Un MBean a pour rôle de fournir une interface qui permet d'obtenir des informations sur la ressource et éventuellement de la gérer au travers de méthodes dédiées.
- agent : le composant principal de ce niveau est un agent qui est un serveur de MBeans dans lequel les MBeans se sont enregistrés. L'agent permet un accès aux MBeans grâce à des connecteurs ou des adapteurs de protocole
- service distribué : fournit une IHM pour permettre d'interagir sur les ressources via l'agent.



L'élément principal du niveau agent est un objet de type « MBean Server » : son rôle est de gérer et de mettre en oeuvre les MBeans qui se sont enregistrés auprès de lui.

Le découpage de l'architecture de l'API en trois couches permet une meilleure répartition des rôles et réduit la complexité des fonctionnalités des différentes couches.

Chacune des trois couches propose des objets avec des interfaces bien définies.

L'élément principal du niveau instrumentation est un objet de type MBean.

La mise en oeuvre d'un MBean standard implique plusieurs étapes :

- définition des fonctionnalités du MBean dans une interface
- créer le MBean qui doit implémenter cette interface
- instancier le MBean
- enregistrer le MBean dans un serveur de MBeans
- utiliser une application de gestion qui va dialoguer avec le serveur de MBeans via un connecteur pour interagir avec le MBean

Cette architecture permet de rendre indépendante la façon dont une ressource est instrumentée de l'infrastructure de gestion utilisée. Cette indépendance est assurée par des connecteurs (connectors) qui permettent à une application de gestion de dialoguer avec un agent JMX. Ce connecteur peut utiliser différents protocoles.

Ceci permet d'intégrer de façon standard la surveillance et la gestion de ressources en Java avec des applications de monitoring et de gestion existantes pour peux que ces applications possèdent un connecteur respectant les spécifications JMX ou qu'il existe un adaptateur pour le protocole utilisé.

31.3. Un premier exemple

Ce premier exemple va utiliser Java SE 5.0 pour créer un MBean de type standard, instancier un serveur de MBean, enregistrer le MBean dans le serveur et interagir avec le MBean grâce à l'outil Jconsole du JDK.

31.3.1. La définition de l'interface et des classes du MBean

Il faut définir l'interface du MBean : son nom doit obligatoire être composé du nom de la classe du MBean suivi de MBean

Exemple :

```
package com.jmdoudoux.tests.jmx;

public interface PremierMBean {

  public String getNom();

  public int getValeur();
  public void setValeur(int valeur);

  public void rafraichir();

}
```

Il faut définir le MBean qui doit implémenter l'interface définie

Exemple :

```
package com.jmdoudoux.tests.jmx;

public class Premier implements PremierMBean {

  private static String nom = "PremierMBean";
  private int valeur = 100;

  public String getNom() {
    return nom;
  }

  public int getValeur() {
    return valeur;
  }

  public synchronized void setValeur(int valeur) {
     this.valeur = valeur;
  }

  public void rafraichir() {
    System.out.println("Rafraichir les donnees");

  }

  public Premier() {
```

```
  }
}
```

Il faut définir une application qui va créer un serveur de MBean, instancier le MBean et l'enregistrer dans le serveur.

Exemple :

```
package com.jmdoudoux.tests.jmx;

import java.lang.management.ManagementFactory;

import javax.management.InstanceAlreadyExistsException;
import javax.management.MBeanRegistrationException;
import javax.management.MBeanServer;
import javax.management.MalformedObjectNameException;
import javax.management.NotCompliantMBeanException;
import javax.management.ObjectName;

public class LancerAgent {

  public static void main(String[] args) {
    MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();

    ObjectName name = null;
    try {
      name = new ObjectName("com.jmdoudoux.tests.jmx:type=PremierMBean");

      Premier mbean = new Premier();

      mbs.registerMBean(mbean, name);

      System.out.println("Lancement ...");
      while (true) {

        Thread.sleep(1000);
        mbean.setValeur(mbean.getValeur() + 1);
      }
    } catch (MalformedObjectNameException e) {
      e.printStackTrace();
    } catch (NullPointerException e) {
      e.printStackTrace();
    } catch (InstanceAlreadyExistsException e) {
      e.printStackTrace();
    } catch (MBeanRegistrationException e) {
      e.printStackTrace();
    } catch (NotCompliantMBeanException e) {
      e.printStackTrace();
    } catch (InterruptedException e) {
    }
  }
}
```

31.3.2. L'exécution de l'application

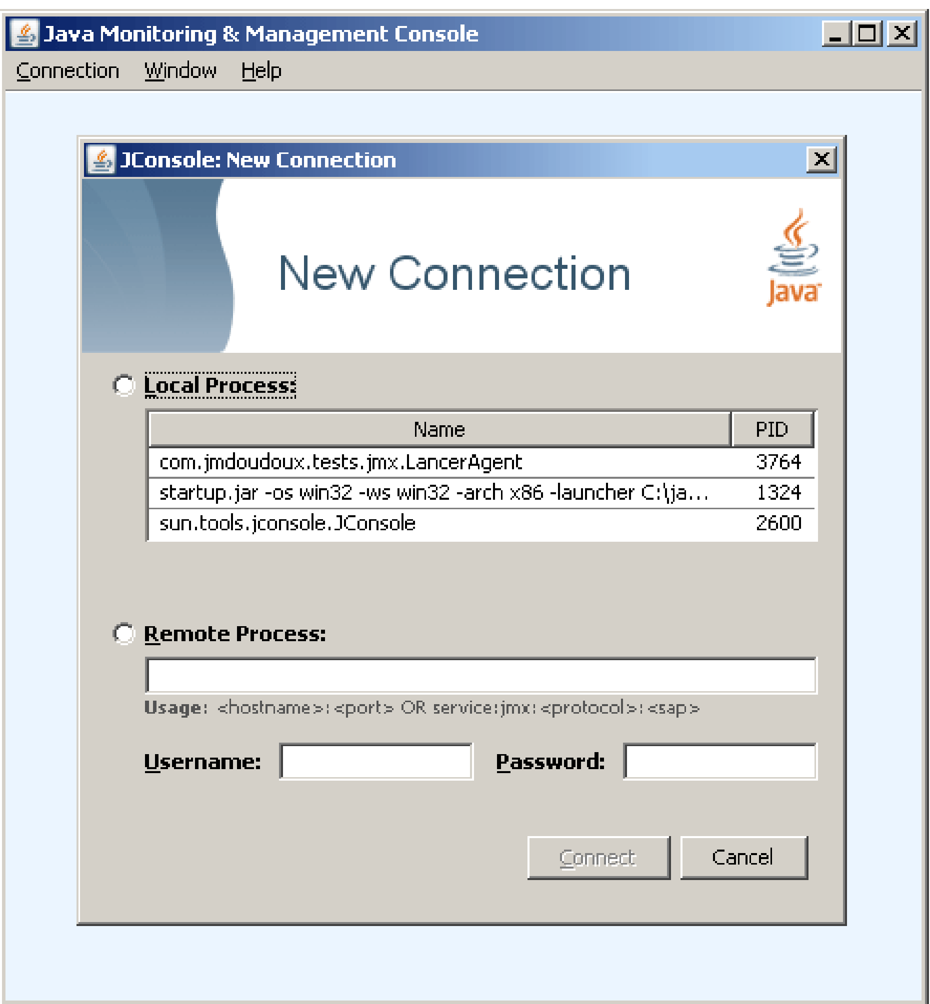
Il faut compiler la classe et l'exécuter en demandant l'activation de l'accès distant aux fonctionnalités de JMX

Exemple :

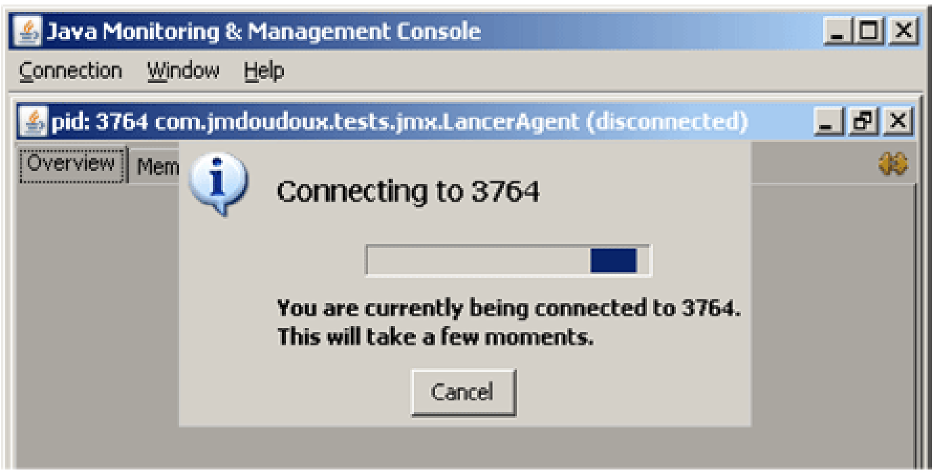
```
java -Dcom.sun.management.jmxremote com.jmdoudoux.tests.jmx.LancerAgent
```

Il est alors possible d'accéder à l'agent en utilisant par exemple l'outil JConsole fourni avec le JDK à partir de sa version 5.0 ou Visual VM fourni avec le JDK à partir de sa version 6.0

Sous Windows, il faut ouvrir une nouvelle boîte de commandes et lancer la commande jconsole du JDK.

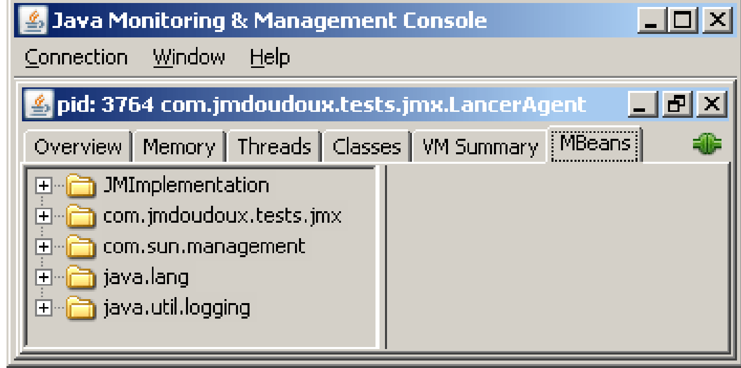


Il faut sélectionner « Local Process » puis la classe dans lequel le serveur MBean est en cours d'exécution et cliquez sur le bouton « Connect »

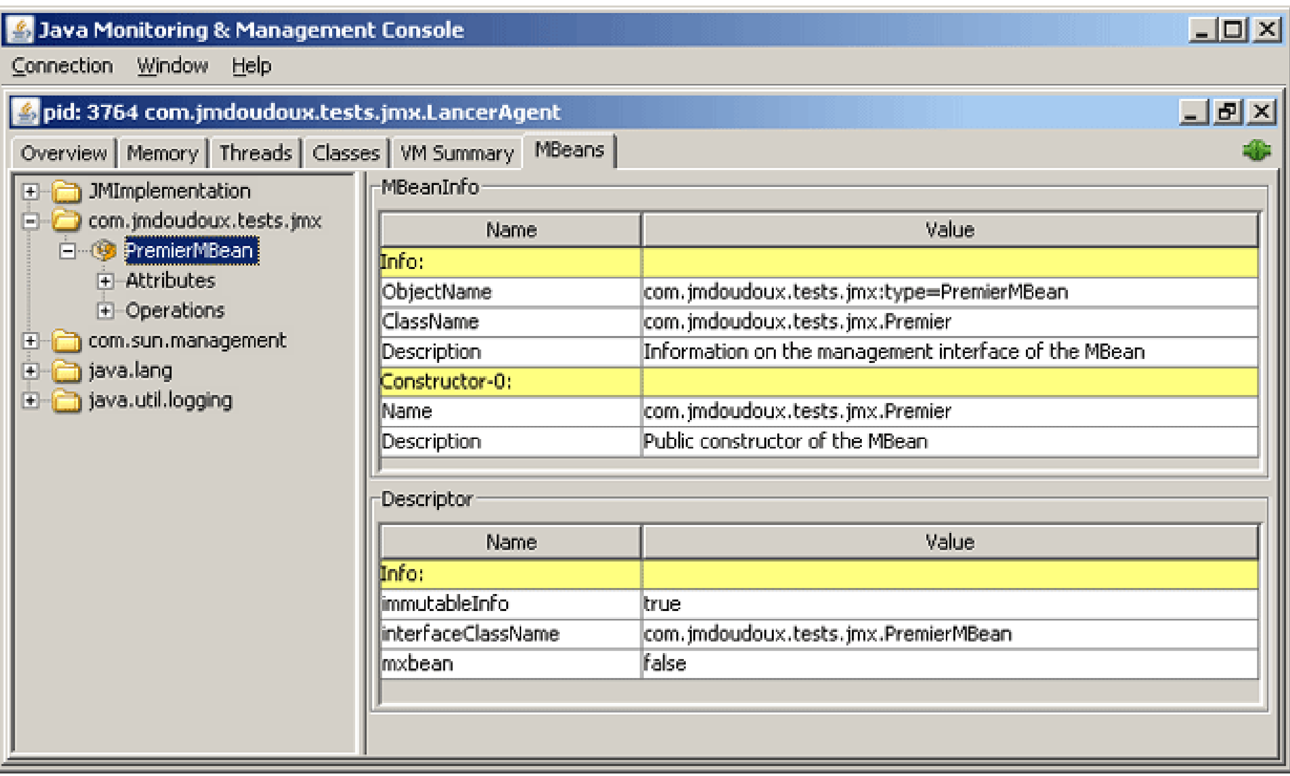


JConsole tente de se connecter au processus de la JVM.

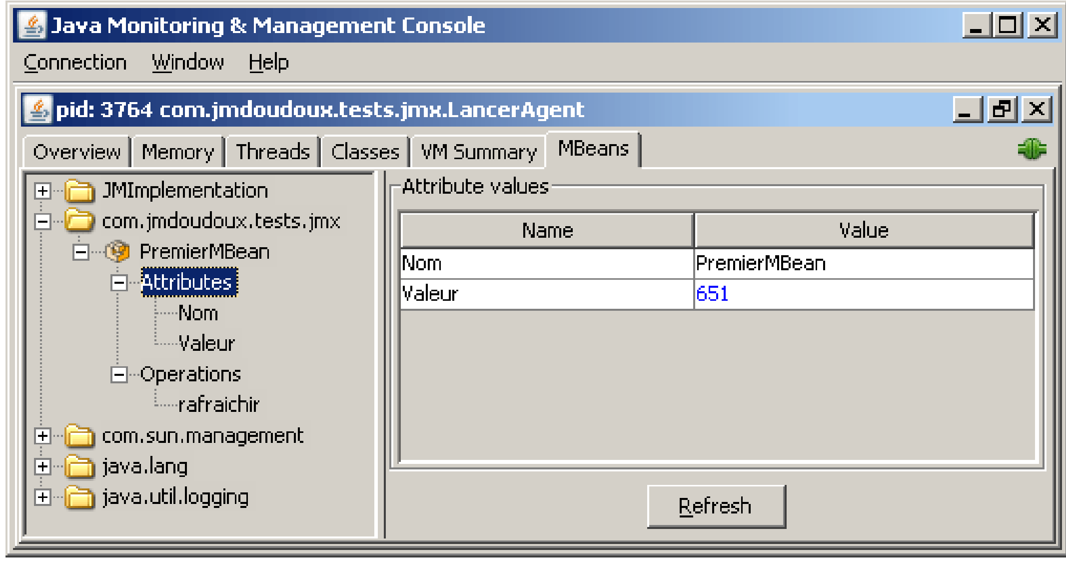
JConsole permet d'obtenir des informations sur la JVM et de gérer les MBeans.



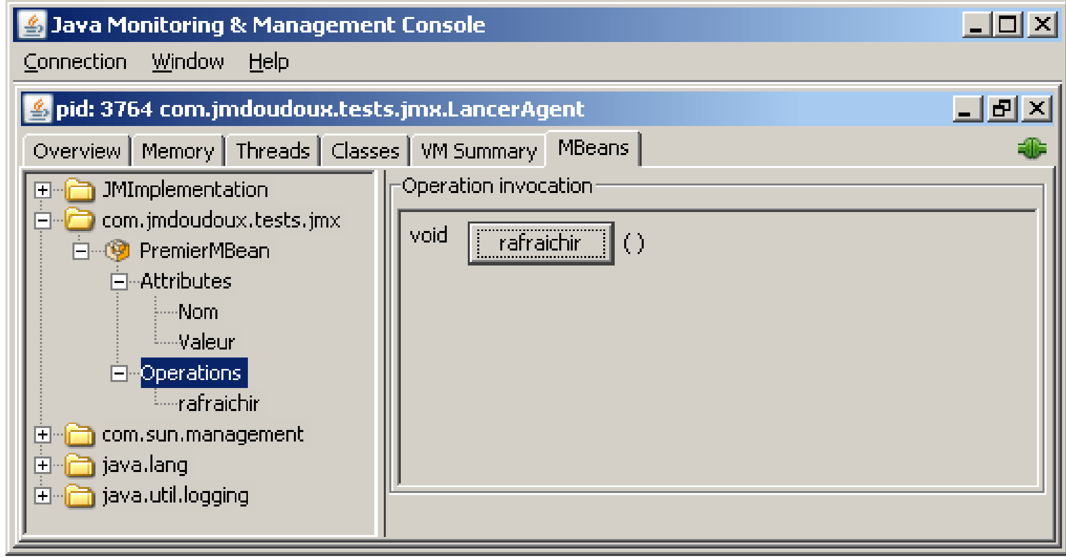
Il faut sélectionner dans l'arborescence le MBean concerné



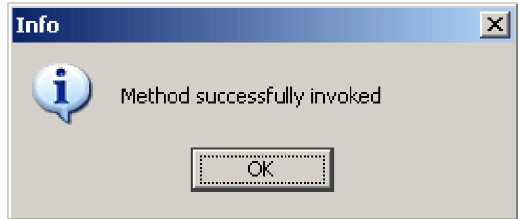
La branche Attributes permet de voir les différents attributs



La branche « Operations » permet d'invoquer les méthodes du MBeans



Il faut sélectionner la méthode et cliquer sur le bouton



31.4. La couche instrumentation : les MBeans

La couche instrumentation assure l'instrumentation de ressources grâce aux MBeans.

L'instrumentation consiste à écrire des MBeans qui vont permettre d'instrumenter des ressources.

Pour être instrumentée, une ressource doit être une classe Java ou être encapsulée dans une classe Java. C'est notamment le cas si la ressource est un appareil. L'instrumentation se fait au moyen de la définition d'une interface qui décrit les fonctionnalités d'instrumentation proposées et une classe de type MBean qui implémente cette interface.

Rien n'est imposé quant à la granularité de la ressource à instrumenter : celle ci peut aller d'une simple classe jusqu'à une application dans son intégralité.

L'instrumentation d'une ressource se fait grâce à un Managed Bean ou MBean. Il existe plusieurs types de MBean.

- MBean standard : c'est une classe Java qui implémente une interface dédiée et respecte les spécifications de JMX.
- MBean dynamique : classe Java encapsulant un MBean qui offre plus de possibilités au runtime pour exposer dynamiquement ses fonctionnalités.

31.4.1. Les MBeans

Un MBean est l'élément de la spécification JMX le plus bas : son rôle est d'assurer la communication avec la ressource à gérer. MBean est l'abréviation de Managed Bean.

Un MBean a pour rôle de permettre la gestion et le dialogue avec une ressource. La nature d'une telle ressource est variée : application, composant, service, dispositif, appareil électronique, etc ...

Pour gérer des ressources via JMX, il faut tout d'abord instrumenter la ressource en développant une classe qui sera le MBean.

Un MBean est une classe Java respectant les spécifications JMX qui implémente une interface particulière. Un MBean possède les caractéristiques suivantes :

- doit être une classe public concrète
- doit avoir au moins un constructeur public
- doit implémenter sa propre interface dont le nom est celui de la classe du MBean suffixé par MBean ou implémenter l'interface DynamicMBean

Un MBean est accessible via son interface qui peut proposer :

- l'appel des constructeurs du MBean
- l'obtention et/ou la modification de la valeur de propriétés (attributs en lecture et/ou écriture)
- l'invocation des méthodes
- l'émission de certaines notifications lorsque certains événements surviennent
- une description pour les fonctionnalités proposées

Remarque : il n'est pas recommandé de surcharger des méthodes exposées par un MBean.

Un MBean est plus qu'une interface puisqu'il doit contenir le code permettant les interactions avec la classe ou l'appareil qu'il doit instrumenter et/ou surveiller.

En plus de l'instrumentation, un MBean peut émettre des notifications en réponse à des événements. Le modèle de notifications proposé par JMX pour les MBeans repose sur le modèle des événements Java. Ces notifications permettent aux MBeans et à l'agent qui le gère de notifier certains événements à un ou plusieurs abonnés.

31.4.2. Les différents types de MBeans

Les MBeans sont répartis en deux grandes familles : standard MBeans et dynamic MBeans.

Il existe quatre types de MBeans plus ou moins complexe à développer :

- MBean standard : ce sont des Java beans qui implémentent une interface définie de façon statique par le développeur : leurs fonctionnalités sont décrites dans cette interface implémentée par le MBean. Dans un tel objet, le nombre de propriétés exposées par le MBean est fixe puisque défini par l'interface. C'est le type le plus utilisé car c'est le plus simple à implémenter.
- MBean dynamic : ils exposent les informations concernant leurs fonctionnalités au travers de méta données. Ils implémentent l'interface DynamicMBean qui leur permet d'exposer leurs fonctionnalités dynamiquement à l'exécution : le nombre de propriétés exposées peut donc être variable. Chaque attribut, opération et notification doit être découvert à l'exécution. Leur écriture est relativement complexe.
- Open MBean : ce sont des MBeans dynamic qui respectent des conventions ce qui les rends un peu plus complexe mais ils sont en contre partie plus portables. Ces conventions imposent notamment de n'utiliser que des types de base de Java et certaines classes définies dans les spécifications JMX. Il est alors inutile de rajouter des classes au classpath de l'agent et des clients.
- Model MBean : ce sont des MBeans dynamic génériques qui peuvent être entièrement configurés. Ils sont fournis par l'implémentation de JMX utilisée et leur mise en oeuvre dépend de cette implémentation.

Les MBeans standard et dynamic publient tous les deux dynamiquement leur interface :

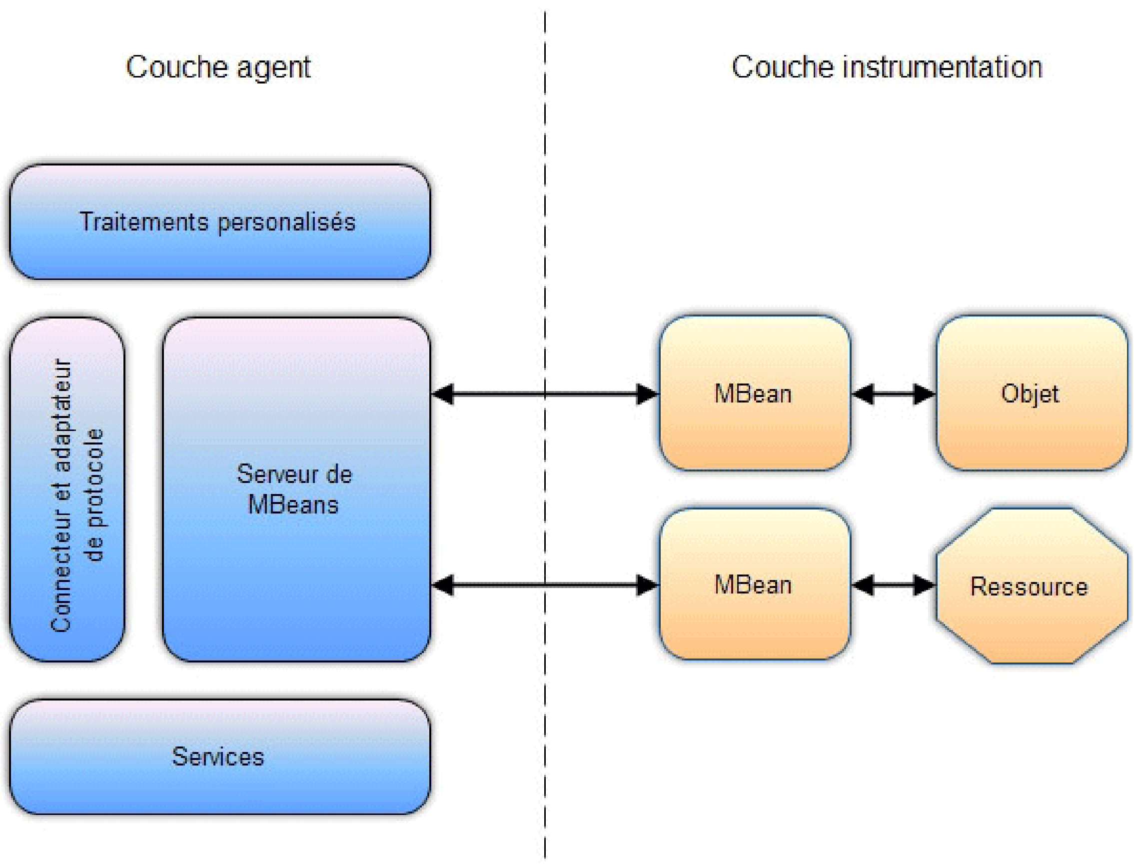
- avec les MBeans dynamic, le développeur doit coder les méthodes pour publier son interface
- avec les MBeans standard, JMX utilise l'introspection pour publier l'interface du MBean

Les MBeans dynamic publient les méthodes de l'interface du MBean en fournissant une description de chacune des méthodes exposées aux clients JMX.

Les MBeans dynamic utilisent des classes qui encapsulent les méta données d'un MBean. Ces méta données permettent de décrire la structure et les fonctionnalités du MBean (constructeurs, attributs, opérations et notifications). Ces méta données comprennent un nom, une description et des caractéristiques.

31.4.3. Les MBeans dans l'architecture JMX

Les composants MBeans sont gérés par un serveur de MBeans.



Chaque MBean enregistré dans le serveur de MBeans d'un agent JMX expose son interface aux applications de gestion. Un MBean est accédé de l'extérieure via l'agent dans lequel il est enregistré.

Lorsque l'on créé un MBean, il n'est donc pas utile de se soucier du type d'agent ou d'application de gestion qui va solliciter le MBean : un MBean n'a donc pas besoin d'avoir de référence sur le serveur qui gère son cycle de vie.

31.4.4. Le nom des MBeans

Chaque MBean possède un ObjectName qui doit être choisi judicieusement car il permet de l'identifier. Un ObjectName est un objet de type javax.management.ObjectName.

Un ObjectName encapsule le nom d'un MBean ou un motif de recherche de noms de MBean.

Un ObjectName est composé de deux parties :

- un nom de domaine qui peut être le domaine par défaut ou un domaine personnalisé
- un ensemble de propriétés sous la forme de paires clé = valeur séparées par une virgule. Au moins une propriété doit être définie

La syntaxe d'un ObjectName est de la forme [nomDomain]:propriete=valeur[,propriete=valeur]*

Le contenu d'un ObjectName devrait permettre de facilement déterminer le rôle du MBean.

Exemple :

com.jmdoudoux.test.jmx:type=MaRessourceBean,name=maRessource

Le domaine est une chaîne de caractères arbitraire facultative. Si le domaine est vide alors cela implique l'utilisation du domaine par défaut du serveur de MBeans dans lequel l'ObjectName est utilisé.

Il est recommandé de préfixer le nom de domaine par le nom du package Java pour éviter les collisions de noms entre MBeans.

Le nom de domaine ne doit pas contenir de caractères / qui est réservé pour les hiérarchies de serveurs de MBean. Le domaine ne peut pas contenir de caractères «:» puisqu'il est utilisé comme séparateur entre le domaine et les propriétés.

Si le domaine contient au moins un caractère «*» ou «?» alors l'ObjectName est un motif pour la recherche de MBeans. Ces deux caractères ne peuvent pas être utilisés dans le nom d'un MBean.

La liste de propriétés doit obligatoirement contenir au moins une propriété sous la forme clé=valeur. Chaque propriété de la liste est séparée par un caractère virgule. Le nom de la propriété doit respecter les conventions de nommage des entités Java.

Chaque ObjectName d'un même type devrait avoir le même ensemble de propriétés. Les valeurs de ces propriétés vont permettre de différencier plusieurs instances d'un même type. Parmi ces propriétés, il est fréquent de trouver la propriété name.

Hormis la clé type, toutes les autres clés peuvent être librement utilisées. Les clés sont généralement utilisées par les clients JMX pour afficher une représentation graphique hiérarchique du MBean.

La JSR 77 définit une propriété j2eeType qui possède un rôle similaire. Les propriétés type et j2eeType peuvent être utilisées simultanément.

Attention : les espaces contenus dans l'ObjectName sont tous significatifs et les ObjectNames sont sensibles à la casse.

L'ordre des propriétés n'est pas significatif.

La valeur d'une propriété peut être entourée de double quotes surtout si elle contient des caractères spéciaux comme le caractère virgule par exemple.

Exemple :

com.jmdoudoux.test.jmx:type=MaRessourceBean,name=maRessource, taille="200"

com.jmdoudoux.test.jmx:type=MaRessourceBean,name=maRessource, taille="200,250"

Le caractère * peut aussi être utilisé dans la liste de propriétés sur l'ObjectName : dans ce cas il concerne un motif de recherche

Exemple :

com.jmdoudoux.test.jmx:type=MaRessourceBean,*

31.4.5. Les types de données dans les MBeans

Les MBeans peuvent être amenés à manipuler des types plus ou moins complexes de données pour différents besoins :

- pour typer un attribut qui est alors utilisé dans les getter et setter
- pour des paramètres des méthodes et leur valeur de retour
- pour lever des exceptions dans les méthodes
- pour les notifications et les données qu'elles utilisent

31.4.5.1. Les types de données complexes

Il est fréquent d'avoir besoin des types de données complexes qui encapsulent les informations d'une entité.

Il est parfois possible si ces données sont des attributs du MBeans de séparer ces données en différents attributs de type communs. Cependant ceci n'est pas possible pour différentes autres situations :

- le type complexe est lui même composé de type complexe
- le type complexe comporte de nombreux attributs et doit être utilisé en paramètre d'une méthode
- le type complexe doit être utilisé comme valeur de retour d'une méthode

L'utilisation de ces types complexes va inévitablement poser des soucis avec certains clients générique comme l'outil JConsole ou l'adaptateur de protocoles HTML car naturellement ces types de données ne sont pas connus.

JMX propose également au travers des spécifications des Open MBeans une solution pour utiliser des types complexes de façon portable.

31.5. Les MBeans standards

Ce sont les plus simples des MBeans. Il suffit de définir une interface dont le nom est composé du nom de la classe du MBean et du suffixe MBean.

Les fonctionnalités de base de JMX sont faciles à mettre en oeuvre dans un MBean standard puisqu'il suffit d'écrire une interface qui décrit les fonctionnalités du MBean et de fournir une implémentation dans une classe qui respecte les conventions Java Bean.

Dans un MBean standard, l'interface du MBean ne peut pas être modifiée à l'exécution : JMX propose d'autres types de MBeans pour adresser ce besoin.

31.5.1. La définition de l'interface d'un MBean standard

Un MBean peut contenir des accesseurs (getters et/ou setters) pour des attributs et des méthodes qui pourront être invoquées pour réaliser des actions.

Ainsi un MBean peut proposer :

- des attributs en lecture et/ou écriture
- des opérations qui pourront être invoquées
- des notifications qui pourront être émises par le MBean et envoyées à des abonnés

Dans un MBean standard, ces différents éléments sont définis de façon statique dans une interface. Cette interface peut donc contenir :

- la définition des getters/setters sur les attributs du MBean
- la définition des opérations proposées par le MBean

Exemple :

```
package com.jmdoudoux.tests.jmx;

public interface PremierMBean {

  public String getNom();

  public int getValeur();
  public void setValeur(int valeur);

  public void rafraichir();

```

```
}
```

Dans un MBean standard, tous les constructeurs public sont exposés automatiquement même le constructeur par défaut si celui ci est créé par le compilateur. Ainsi un client JMX peut instancier un MBean en utilisant un des constructeurs public.

Les méthodes d'un MBean peuvent être des getters et setters sur des attributs ou des opérations qui permettront de réaliser certains traitements. Les méthodes de type getter et setters doivent respecter les conventions de nommage des Java beans. Il n'est pas possible de surcharger un getter ou un setter.

Si les conventions de nommage des Java beans ne sont pas respectées ou si une incohérence est détectée entre le type utilisé dans le getter et le setter, une exception sera levée lors de l'utilisation du MBean.

Exemple :

```
package com.jmdoudoux.tests.jmx;

import java.io.IOException;

public interface SecondMBean {

  public int getValeur() throws IOException;
  public void setValeur(String valeur) throws IOException;

}
```

Lors d'une tentative d'enregistrement d'un MBean implémentant cette interface, une exception de type NotCompliantException est levée.

Exemple :

```
javax.management.NotCompliantMBeanException: Getter and setter for Valeur have inconsistent
types
```

Chaque méthode publique qui n'est pas identifiée comme étant un getter ou un setter d'une propriété est considérée comme une opération exposée par le MBean. Une opération peut avoir un nombre quelconque de paramètres et éventuellement avoir une valeur de retour.

Les spécifications JMX préconisent de ne pas surcharger les méthodes exposées des MBeans.

Une méthode d'un MBean peut lever des exceptions qui devront être gérées par le client JMX. Il est recommandé pour un MBean de ne lever que des exceptions fournies en standard par la plate-forme Java SE. Si un MBean lève une exception non standard, l'application qui utiliserait ce MBean lèvera une exception de type ClassNotFoundException.

C'est une best practice que chaque méthode déclarée dans l'interface d'un bean standard indique qu'elle peut lever l'exception java.io.IOException. Ceci impose la prise en compte de cette exception notamment lors de l'utilisation d'un proxy sinon une erreur de communication lèvera une exception de type UndeclaredThrowableException qui encapsulera l'exception de type IOException.

Exemple :

```
package com.jmdoudoux.tests.jmx;

import java.io.IOException;

public interface PremierMBean {

  public String getNom() throws IOException;

  public int getValeur() throws IOException;
  public void setValeur(int valeur) throws IOException;

```

```
  public void rafraichir() throws IOException;

}
```

Si le MBean n'est accédé que dans la JVM dans laquelle elle s'exécute, il n'est pas utile de déclarer qu'une méthode peut lever une exception de type IOException. Mais généralement, l'accès aux MBeans se fait de façon distante. Pour simplifier l'utilisation locale, il est possible de définir une interface fille qui redéfinit les méthodes mais sans déclarer la levée de l'exception.

Exemple :

```
package com.jmdoudoux.tests.jmx;

public interface PremierLocalMBean {

  public String getNom();

  public int getValeur();
  public void setValeur(int valeur);

  public void rafraichir();

}
```

31.5.2. L'implémentation du MBean Standard

La classe du MBean doit implémenter l'interface du MBean. Le nom de la classe doit obligatoirement être identique à celui de l'interface sans le suffixe MBean. Si ce n'est pas le cas, une exception de type javax.management.NotCompliantMBeanException est levée.

Seules les propriétés et opérations définies dans l'interface seront utilisables par l'agent JMX. L'agent JMX va utiliser l'introspection pour déterminer la liste des propriétés et des opérations supportées par le MBean en recherchant l'interface du MBean.

Pour transformer une classe Xyz en MBean standard, il faut :

- créer une interface XyzMBean qui contient les méthodes et les getter et setter des attributs à exposer
- la classe Xyz doit implémenter l'interface XyZMBean

Afin d'assurer une séparation des rôles, il est préférable de séparer dans deux classes distinctes la ressource gérée ou l'encapsulation de cette ressource et la classe du MBean qui va gérer la ressource. Il suffit pour cela que le MBean possède une référence sur la classe de la ressource, généralement passée dans le constructeur du MBean.

31.5.3. L'utilisation d'un MBean

Chaque MBean doit être enregistré dans un serveur de MBeans avec un identifiant unique sous la forme d'un nom d'objet (ObjectName).

Un ObjectName est composé d'un nom de domaine et d'attributs sous la forme de paires clé/valeur. La combinaison du nom de domaine et des attributs doit obligatoirement être unique dans un serveur de MBeans.

Les méthodes déclarées dans l'interface du MBean seront accessibles aux clients JMX.

Les constructeurs public sont toujours accessibles aux clients JMX par introspection.

Les propriétés accessibles aux clients JMX sont exposées grâce aux getters et setters définies dans l'interface du MBean.

Il n'y a rien qui peut empêcher l'utilisation des MBeans directement sans passer par JMX puisque ce sont de simples Java beans.

31.6. La couche agent

Si une ressource est instrumentée par un MBean, alors la gestion du MBean est assurée par un agent JMX : il assure donc la gestion et l'exploitation de différents MBeans.

Le niveau agent est essentiellement composé d'un agent JMX qui possède plusieurs responsabilités :

- instancier et gérer les MBeans dans un serveur de MBeans
- charger et initialiser le ou les connecteurs et adaptateurs de protocoles pour dialoguer avec des clients
- fournir des services définis dans les spécifications JMX

Le composant principal d'un agent JMX est un serveur de MBeans. Un serveur de MBeans enregistre et gère des MBeans. Généralement un agent JMX s'exécute dans la JVM où s'exécute les MBeans qu'il gère mais ce n'est pas une obligation. L'agent permet à une application d'interagir avec les MBeans par son intermédiaire en utilisant des connecteurs ou des adaptateurs de protocoles.

Un serveur de MBeans est un registre pour MBeans : il gère le cycle de vie des MBeans qui s'enregistrent auprès de lui. Le serveur de MBeans permet un accès aux MBeans à des applications tierces en exposant leurs interfaces.

Les MBeans peuvent aussi être instanciés et enregistrés dans le serveur de MBean par un autre MBean ou par l'agent. Chaque MBean s'enregistre avec un identifiant unique de type ObjectName. Cet identifiant est fourni par le développeur, c'est donc lui qui doit être le garant de son unicité dans un même serveur de MBeans.

Un agent JMX permet donc à une application de gestion d'invoquer les fonctionnalités des MBeans : il assure la communication entres les MBeans et les interfaces de gestions grâce à plusieurs entités : un serveur de MBeans et un ou plusieurs adaptateurs de protocoles ou connecteurs.

Par défaut un agent ne possède pas de possibilités de communications. Les connecteurs et adaptateurs de protocoles permettent à un agent de communiquer en utilisant un protocole tel que HTTP, SNMP, ...

L'implémentation par défaut propose un adaptateur de protocoles pour HTML, qui permet de disposer, pour n'importe quel agent JMX, d'une console d'administration accessible depuis un navigateur internet. C'est sur ce principe que fonctionne la console JMX fournit en standard avec JBoss par exemple.

Un agent JMX peut se voir ajouter des fonctionnalités dynamiquement sous la forme de services. Plusieurs services sont fournis en standard et il est possible de développer ces propres services. Généralement, ils sont fournis sous la forme de MBean et sont donc administrables via JMX et le serveur de MBeans qui les gère.

31.6.1. Le rôle d'un agent JMX

Un agent JMX sert d'intermédiaire entre les MBeans et un client JMX (généralement une application de gestion) : il assure l'indépendance entre la ressource gérée et l'application de gestion distante.

Pour gérer le MBean et permettre son accès, il faut l'enregistrer dans un agent JMX. C'est le serveur de MBeans qui est le composant principal de l'agent qui assure la gestion du cycle de vie des MBeans qui se sont enregistrés auprès de lui.

L'agent JMX va utiliser l'introspection sur l'interface pour déterminer les fonctionnalités offertes par un MBean standard.

La communication entre ce serveur et les applications clientes se fait via des adaptateurs de protocoles ou des connecteurs qui assurent la communication de façon indépendante du MBean.

L'agent JMX propose aussi plusieurs services offrant différentes fonctionnalités.

Généralement, un agent JMX s'exécute dans la JVM ou sont exécutés les MBeans mais ce n'est pas une obligation.

31.6.2. Le serveur de MBeans (MBean Server)

Le MBean Server compose le coeur de l'agent : il gère les MBeans qui se sont enregistrés auprès de lui grâce à un identifiant unique (Object Name). Le serveur de MBeans est alors en charge de la gestion de ces MBeans.

Ce serveur permet de gérer le cycle de vie des MBeans (ajout, modification ou suppression) et de permettre leur utilisation de manière locale ou distante. C'est le coeur du système de gestion proposé par JMX.

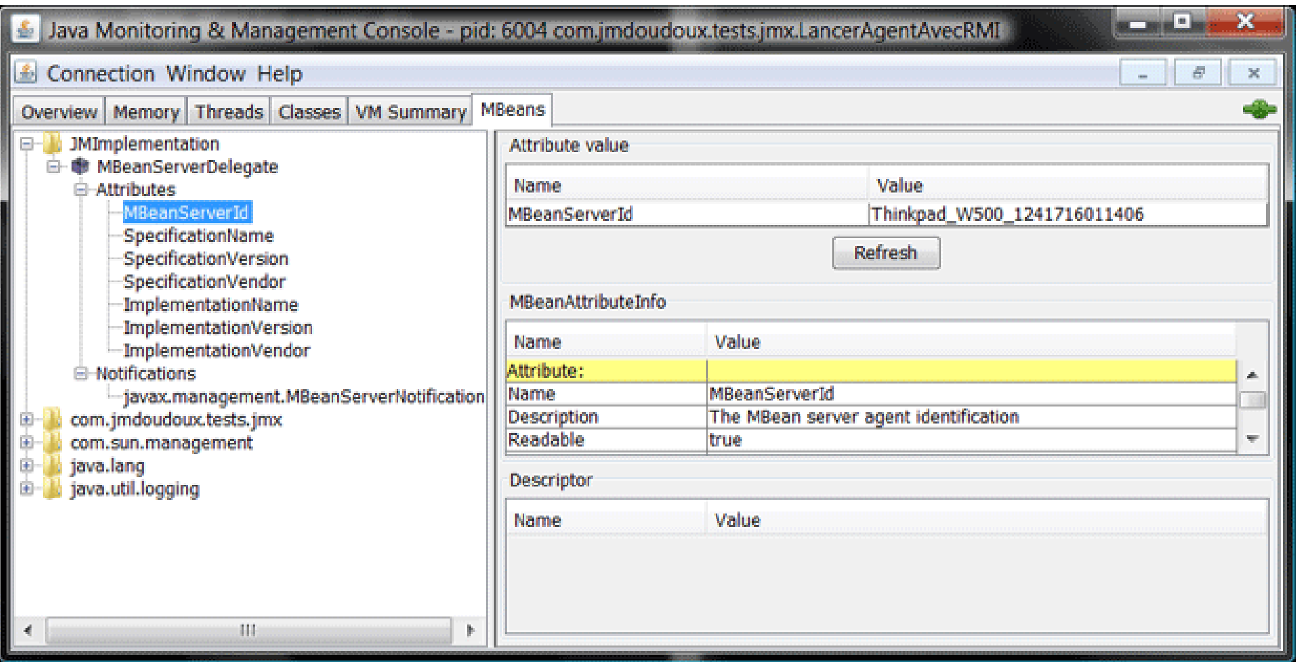
Un serveur de MBeans possède plusieurs fonctionnalités notamment :

- agir comme un registre pour les MBeans qui se sont enregistrés auprès de lui
- gérer le cycle de vie des MBeans
- découvrir et exposer les fonctionnalités des interfaces des MBeans aux applications de gestion
- lire et modifier les valeurs des propriétés d'un MBean
- invoquer les méthodes du MBean
- obtenir les notifications émises par le MBean

31.6.3. Le Mbean de type MBeanServerDelegate

Lorsqu'un serveur de Mbeans est instancié, un MBean de type MBeanServerDelegate est automatiquement instancié et enregistré dans le serveur avec un ObjectName qui vaut « JMImplementation:type=MBeanServerDelegate ».

Cet MBean permet d'obtenir des informations sur le serveur MBean sous la forme de plusieurs attributs en lecture seule : MBeanServerId, SpecificationName, SpecificationVersion, SpecificationVendor, ImplementationName, ImplementationVersion et ImplementationVendor.



Le plus utile de ces attributs est MBeanServerId puisqu'il fournit l'identifiant du serveur de MBeans.

C'est aussi lui qui est responsable des notifications de type jmx.mbean.created et jmx.mbean.deleted émises par le serveur de MBeans notamment lors de l'enregistrement ou la suppression de MBeans du serveur.

31.6.3.1. L'enregistrement d'un MBean dans le serveur de MBeans

Chaque MBean doit être associé à une instance d'un objet de type ObjectName lors de l'enregistrement dans le serveur de MBeans. Un objet de type ObjectName sert d'identifiant unique et doit respecter les spécifications JMX :

- il doit avoir un domaine (généralement le package qui contient la classe du MBean) ou à défaut le domaine par défaut du serveur
- un ensemble de propriétés (exemple le type d'objet)

Pour utiliser un MBean, il faut donc l'enregistrer dans le serveur de MBeans d'un agent JMX. Le plus simple pour réaliser cette opération est de suivre deux étapes :

- obtenir le serveur de MBeans de la JVM : le plus facile est d'utiliser la méthode getPlatformMBeanServer() de la classe ManagementFactory qui permet d'obtenir un serveur de MBeans en cours d'exécution ou une nouvelle instance d'un serveur de MBeans si aucun n'est déjà en cours d'exécution dans la JVM.
- enregistrer le MBean auprès de l'instance du serveur obtenue : le MBean est enregistré dans le serveur de MBeans en utilisant la méthode registerMBean() qui attend en paramètre une instance du MBean et l'objet de type ObjectName associé au MBean.

31.6.3.2. L'interface MBeanRegistration

L'interface javax.management.MBeanRegistration peut être implémentée par un MBean pour définir des callbacks qui seront invoqués par le serveur de MBeans durant le cycle de vie du MBean. Ces callbacks fournissent un mécanisme de contrôle sur le processus d'enregistrement et de désenregistrement du MBean.

Elle définit quatre méthodes :

Méthode	Rôle
ObjectName preRegister(MBeanServer mbs, ObjectName name)	Invoqués juste avant que le MBean ne soit enregistré du serveur de MBeans
void postRegister()	Invoqués juste après que le MBean soit correctement enregistré du serveur de MBeans
void preDeRegister()	Invoqués juste avant que le MBean ne soit désenregistré du serveur de MBeans
void postDeRegister()	Invoqués juste après que le MBean soit correctement désenregistré du serveur de MBeans

31.6.3.3. La suppression d'un MBean du serveur de MBeans

La méthode unregisterMBean() permet de supprimer un MBean du serveur de MBeans : elle attend en paramètre l'identifiant du Mbean sous la forme de son ObjectName

Une fois cette méthode invoqué, le serveur ne possède plus de référence sur l'instance du MBean.

Remarque : la destruction d'un serveur de MBeans entraîne la destruction des MBeans dans lequel ils étaient enregistrés.

31.6.4. La communication avec la couche agent

Les agents ne communiquent pas directement avec la couche services distribués : cette communication est assurée via des connecteurs et/ou des adaptateurs de protocoles. Ceci permet d'utiliser plusieurs protocoles comme HTTP ou SNMP.

Les adaptateurs de protocoles permettent d'accéder à un agent JMX en utilisant un protocole donné : ils adaptent les échanges avec l'agent en utilisant le protocole pour lequel ils ont été écrit

Généralement les adaptateurs de protocoles ont uniquement une partie serveur qui se charge d'adapter les échanges au format du protocole utilisé.

Un connecteur permet un échange entre un client JMX et un agent JMX en utilisant un protocole particulier. Ces échanges sont assurés par une partie du connecteur côté client et une autre côté serveur.

Un connecteur implique obligatoirement une partie côté client et une partie sur l'agent.

L'implémentation de référence propose un adaptateur de protocoles pour HTML qui permet d'avoir un accès à un agent JMX avec un simple navigateur (attention : ce connecteur n'est pas fournie avec le JDK).

La mise en oeuvre des connecteurs et des adaptateurs de protocoles est détaillées dans une des sections suivante.

31.6.5. Le développement d'un agent JMX

Le développement d'un agent comporte plusieurs étapes :

- Instancier un serveur de MBeans
- Démarrer le serveur
- Instancier et enregistrer le ou les MBeans dans le serveur
- Instancier un connecteur ou un adaptateur de protocoles
- Démarrer le connecteur ou l'adaptateur de protocoles
- Eventuellement instancier et enregistrer les services de l'agent

Remarque : généralement les services, les connecteurs et les adaptateurs de protocoles sont implémentés sous la forme de MBeans qu'il est possible d'enregistrer dans le serveur de MBeans pour permettre leur administration.

31.6.5.1. L'instanciation d'un serveur de MBeans

Pour instancier un serveur de MBeans, il faut utiliser directement ou indirectement une fabrique de type MBeanServerFactory. Cette fabrique ne possède aucune instance car toutes les méthodes qu'elle propose sont statiques.

La fabrique peut conserver en interne des références sur les instances de type MBeanServer qu'elle créé.

Elle propose plusieurs méthodes pour obtenir ou manipuler une instance de type MBeanServer notamment :

Méthode	Rôle
MBeanServer createMBeanServer()	Renvoyer une nouvelle instance d'un objet qui implémente l'interface MBeanServer associée au nom de domaine par défaut (DefaultDomain).
MBeanServer createMBeanServer(String domain)	Renvoyer une nouvelle instance d'un objet qui implémente l'interface MBeanServer associée au nom de domaine fourni en paramètre.
ArrayList<MBeanServer> findMBeanServer(String agentId)	Retourner une collection des instances de MBeanServer instanciées avec la méthode createMBeanServer() toujours présent dans la fabrique. Toutes les instances sont retournées avec null en paramètres.
MBeanServer newMBeanServer()	Renvoyer une nouvelle instance d'un objet qui implémente l'interface MBeanServer associée au nom de domaine par défaut (DefaultDomain) sans conserver de référence sur l'instance
MBeanServer newMBeanServer(String domain)	Renvoyer une nouvelle instance d'un objet qui implémente l'interface MBeanServer associée au nom de domaine fourni en paramètre sans conserver de référence sur l'instance

void releaseMBeanServer(MBeanServer mbeanServer)	Supprimer les références au MBeanServer dans la fabrique pour permettre au ramasse miette de libérer l'espace mémoire de l'instance

Le paramètre domain attendu par certaines de ces méthodes permet de préciser le domaine utilisé par le serveur de MBeans. Le domaine par défaut est DefaultDomain.

Le plus simple pour obtenir une instance d'un serveur de MBeans est d'invoquer la méthode createMBeanServer() de la classe MBeanServerFactory.

Exemple :

```
...
    MBeanServer mbs = MBeanServerFactory.createMBeanServer();
...
```

La fabrique fait appel à un objet de type MBeanServerBuilder dont une implémentation par défaut est fournie.

Depuis la version 1.2 de JMX, il est possible de remplacer l'implémentation par défaut de la classe MBeanServer en définissant une classe qui héritent de la classe MBeanServerBuilder et qui possède une constructeur par défaut. Il suffit alors de passer le nom pleinement qualifié de cette classe comme valeur de la propriété javax.management.builder.initial de la JVM. Cette fonctionnalité est cependant à réserver pour des besoins très particuliers.

Il est aussi possible d'obtenir une instance de type MBeanServer en utilisant la méthode getPlatformMBeanServer() de la fabrique java.lang.management.ManagementFactory. Cette fabrique permet d'obtenir des instances des MBeans de la JVM et une instance du MBeanServer par défaut de la JVM.

Exemple :

```
...
MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();
...
```

La classe MBeanServerFactory possède la méthode findMBeanServer(String) qui permet de rechercher un ou plusieurs instances de serveurs de MBeans. Elle retourne une collection qui contient les instances de serveurs de MBeans qui correspondent à l'identifiant fournit en paramètre ou à tous les serveurs de MBeans si le paramètre fourni est null.

31.6.5.2. L'instanciation et l'enregistrement d'un MBean dans le serveur

L'interface MBeanServer propose deux méthodes pour enregistrer un MBean :

- ObjectInstance registerMBean(Object, ObjectName) : enregistrer une instance d'un MBean
- ObjectInstance createMBean(String, ObjectName)

Pour utiliser la méthode registerMbean(), il faut créer une instance de type ObjectName qui va encapsuler le nom unique du MBean dans le serveur.

Il faut ensuite créer une instance du MBean et enregistrer le MBean dans le serveur en utilisant la méthode registrerMBean() de l'interface MBeanServer qui attend en paramètre l'instance du MBean et son ObjectName.

Exemple :

```
...
    ObjectName name = null;
    try {
      name = new ObjectName("com.jmdoudoux.tests.jmx:type=PremierMBean");
      Premier mbean = new Premier();
      mbs.registerMBean(mbean, name);
```

```
    } catch (MalformedObjectNameException e) {
      e.printStackTrace();
    } catch (NullPointerException e) {
      e.printStackTrace();
    } catch (InstanceAlreadyExistsException e) {
      e.printStackTrace();
    } catch (MBeanRegistrationException e) {
      e.printStackTrace();
    } catch (NotCompliantMBeanException e) {
      e.printStackTrace();
    }
...
```

La méthode createMBean() possède plusieurs surcharges : elles permettent toutes avec différents paramètres d'instancier dynamiquement un MBean en utilisant l'introspection. Ces paramètres contiennent toujours le nom de la classe de type String et l'ObjectName du MBean. Les autres paramètres permettent de préciser le classloader à utiliser et les paramètres à fournir lors de l'invocation du constructeur du MBean.

Il est possible à plusieurs instances d'un même MBean dans un serveur de Mbeans en leur affectant à chacun un ObjectName distinct.

31.6.5.3. L'ajout d'un connecteur ou d'un adaptateur de protocoles

Pour pouvoir être utilisé par un client JMX, l'agent doit avoir au moins un connecteur ou un adaptateur de protocoles.

Généralement l'un ou l'autre sont fournis sous la forme d'un MBean qu'il convient d'instancier et d'enregistrer auprès du serveur de MBeans.

Il faut enfin les démarrer pour qu'ils commencent à écouter les appels des clients généralement en invoquant leur méthode start().

Le détails de la mise en oeuvre d'un connecteur et d'un adaptateur de protocoles est proposé dans une prochaine section.

31.6.5.4. L'utilisation d'un service de l'agent

Le détails de la mise en oeuvre des services offerts par un agent est proposé dans la prochaine section.

31.7. Les services d'un agent JMX

Un agent JMX propose plusieurs services définis dans la version 1.1 des spécifications JMX visant à rendre la solution de gestion plus riche en fonctionnalités avancées :

- management applet (m-let) : permet le chargement et l'instanciation dynamique de classes en utilisant une url dédiée qui pointe sur un fichier utilisant des tags particuliers pour décrire les MBeans à traiter
- moniteur : permet d'observer les modifications de valeurs de propriétés numérique ou chaîne de caractères d'un MBean pour notifier ces changements à des abonnés
- timer : permet l'envoi de notifications répétitives ou programmées selon une valeur temporelle à des abonnés pour permettre l'exécution de traitements
- relations entre MBeans : permet de définir et de maintenir des associations entre MBeans et d'assurer l'intégrité de ces relations

Ces services peuvent être implémentés sous la forme de MBeans ce qui leur permet d'être utilisés par les autres MBeans et d'être administrables.

31.7.1. Le service de type M-Let

M-Let est l'abréviation de management applet. Le service de type M-Let permet de charger un MBean local ou distant, de l'instancier et de l'enregistrer dans le serveur de MBeans. La description des MBeans à traiter est contenu dans un fichier texte possédant une syntaxe dédiée. Le fichier est fourni au service grâce à une url pointant sur un fichier local ou distant qui contient la définition des MBeans.

Le fichier doit être un fichier texte dans lequel chaque MBean doit être défini avec un tag <MLET>. Ce tag possède plusieurs attributs qui permettent de fournir les informations concernant le MBean.

- Le service M-Let est enregistré en tant que MBean dans le serveur de MBeans.
- Le service M-Let lit le fichier de description précisé par une url. Chaque MBean décrit dans le fichier est traité par le service :
- Chargement de la classe du MBean
- Création d'une instance du MBean
- Enregistrement du MBean dans le serveur de MBeans de l'agent du service

Ce service permet de créer dynamiquement des agents extensibles.

31.7.1.1. Le format du fichier de définitions

Chaque MBean devant être traité par le service M-Let doit avoir une définition dans le fichier sous la forme d'un tag <MLET>

Le format du tag MLET est le suivant :

```
<MLET

    CODE = class | OBJECT = serfile
    ARCHIVE = "archiveList"
    [CODEBASE = codebaseURL]
    [NAME = mbeanname]
    [VERSION = version]
    >
    [arglist]
</MLET>
```

Les attributs du tags MLET sont :

Attribut	Rôle
CODE	Préciser le nom pleinement qualifié de la classe du MBean. La classe doit être présente dans un des fichiers jar précisé via l'attribut ARCHIVE
OBJECT	Préciser un fichier .ser qui contient le résultat de la sérialisation de l'instance du MBean. Ce fichier doit être présent dans un des fichiers jar précisé via l'attribut ARCHIVE L'utilisation de CODE et de OBJECT est exclusive.
ARCHIVE	Préciser un ou plusieurs fichiers jar contenant le ou les MBeans et leurs dépendances. Si plusieurs sont précisés, la valeur de l'attribut doit être entourée de double quotes et chaque jar doit être séparés avec un caractère virgule. Tous les jars utilisés doivent être stockés dans le répertoire précisé par l'attribut CODEBASE . (obligatoire)
CODEBASE	Préciser le répertoire dans lequel les jar sont stockées. L'utilisation de cet attribut n'est obligatoire que si les jar ne sont pas stockés dans le même répertoire que le fichier de description
NAME	Préciser l'ObjectName du MBean lors de son enregistrement dans le serveur de MBeans. Si la valeur de l'attribut commence par un caractère « : » lors l'ObjectName sera préfixé par le nom de domaine par défaut du serveur de MBeans
VERSION	Préciser le numéro de version du MBean et du jar qui le contient

Deux attributs sont obligatoires :

* CODE ou OBJECT : pour préciser le nom de classe ou le fichier .ser qui contient le résultat de la sérialisation du MBean. CODE et OBJECT sont mutuellement exclusifs.
* ARCHIVE pour préciser le ou les jars contenant les classes requises

Le tag MLET possède le tag fils <ARG> qui permet de préciser des arguments qui seront passés au constructeur du MBean lors de son instanciation.

Le tag <ARG> possède deux attributs :

Attribut	Rôle
TYPE	Préciser le type de l'argument. Seuls quelques types possédant une représentation sous la forme d'une chaîne de caractères peuvent être utilisés : java.lang.Boolean, java.lang.Byte, java.lang.Short, java.lang.Long, java.lang.Integer, java.lang.Float, java.lang.Double, java.lang.String
VALUE	Préciser la valeur de l'argument sous la forme d'une chaîne de caractères

Lors de l'instanciation du MBean, le service M-Let va rechercher un constructeur du MBean dont la signature corresponde aux arguments précisés par les tags <ARG>.

Le fichier peut contenir plusieurs tags <MLET>, un pour chaque MBeans qui devront être instanciés et enregistrés dans le serveur de MBeans.

31.7.1.2. L'instanciation et l'utilisation d'un service M-Let dans un agent

La classe javax.management.loading.MLet est une implémentation du service M-Let fournie en standard. L'implémentation d'un service M-Let doit implémenter l'interface javax.management.loading.MLetMBean.

La classe MLet hérite de la classe URLClassLoader ce qui lui permet de télécharger des classes à travers le réseau.

C'est un MBean qui doit être instancié et enregistré dans le serveur de MBean : il est ainsi possible d'utiliser ce MBean à distance en passant par l'agent JMX.

La classe MLet propose la méthode getMbeansFromURL() qui attend en paramètre l'url du fichier de description et qui permet de lire le fichier et de traiter les MBeans qu'il contient. Deux surcharges permettent de préciser l'url sous la forme d'une chaîne de caractères ou d'un objet de type java.net.URL.

31.7.1.3. Un exemple de mise en oeuvre du service M-Let

Il faut développer un agent qui va instancier et enregistrer un objet de type MLet.

L'instance de cet objet va lire un fichier de description qui va permettre d'instancier et d'enregistrer un MBean dans le serveur de MBeans.

Exemple :

```
package com.jmdoudoux.tests.jmx;

import java.lang.management.ManagementFactory;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.Arrays;
import java.util.Set;

import javax.management.Attribute;
import javax.management.AttributeNotFoundException;
import javax.management.InstanceAlreadyExistsException;
```

```
import javax.management.InstanceNotFoundException;
import javax.management.InvalidAttributeValueException;
import javax.management.MBeanException;
import javax.management.MBeanRegistrationException;
import javax.management.MBeanServer;
import javax.management.MalformedObjectNameException;
import javax.management.NotCompliantMBeanException;
import javax.management.ObjectName;
import javax.management.ReflectionException;
import javax.management.ServiceNotFoundException;
import javax.management.loading.MLet;

public class LancerAgentAvecMLet {

  public static void main(String[] args) {
    MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();

    ObjectName name = null;
    try {
      name = new ObjectName("com.jmdoudoux.tests.jmx:type=PremierMBean");

      // Instanciation et enregistrement du Service MLet
      System.out.println("Instanciation et enregistrement du Service MLet");
      MLet mlet = new MLet();
      mlet.setLibraryDirectory("c:/temp");
      mbs.registerMBean(mlet, new ObjectName("Services:type=MLet"));

      // Lecture du fichier de configuration pour instanciation et
      // enregistrement du MBean
      System.out.println("\nLecture du fichier de configuration");
      Set<Object> mbeans = mlet.getMBeansFromURL(new URL(
          "http://localhost:8080/jmx/mlet.txt"));
      for (Object obj : mbeans) {
        System.out.println("Object = " + obj);
      }

      System.out.println("\nClasspath du service MLet : "
          + Arrays.asList(mlet.getURLs()));

      System.out.println("\nRecherche du mbean enregistré");
      Set<ObjectName> names = mbs.queryNames(name, null);
      for (ObjectName objName : names) {
        System.out.println("ObjectName=" + objName);
      }

      System.out.println("\nExecution de l'agent ...");
      while (true) {

        try {
          Thread.sleep(1000);
        } catch (InterruptedException e) {
        }

        int valeur = Integer.valueOf(mbs.getAttribute(name, "Valeur")
            .toString());
        Attribute attr = new Attribute("Valeur", valeur + 1);
        mbs.setAttribute(name, attr);
      }
    } catch (MalformedObjectNameException e) {
      e.printStackTrace();
    } catch (NullPointerException e) {
      e.printStackTrace();
    } catch (InstanceAlreadyExistsException e) {
      e.printStackTrace();
    } catch (MBeanRegistrationException e) {
      e.printStackTrace();
    } catch (NotCompliantMBeanException e) {
      e.printStackTrace();
    } catch (MalformedURLException e) {
      e.printStackTrace();
    } catch (NumberFormatException e) {
      e.printStackTrace();
    } catch (AttributeNotFoundException e) {
      e.printStackTrace();
```

```
    } catch (InstanceNotFoundException e) {
      e.printStackTrace();
    } catch (MBeanException e) {
      e.printStackTrace();
    } catch (ReflectionException e) {
      e.printStackTrace();
    } catch (InvalidAttributeValueException e) {
      e.printStackTrace();
    } catch (ServiceNotFoundException e) {
      e.printStackTrace();
    }
  }
}
```

Il faut rédiger le fichier de description.

Exemple :

```
<MLET CODE=com.jmdoudoux.tests.jmx.Premier
 ARCHIVE="TestJMX.jar"
 CODEBASE=http://localhost:8080/jmx/
 NAME=com.jmdoudoux.tests.jmx:type=PremierMBean></MLET>
```

Il faut un serveur web sur lequel il faut mettre :

- le fichier mlet.txt
- un fichier jar qui contienne la classe du MBean (TestJMX.jar dans cet exemple)

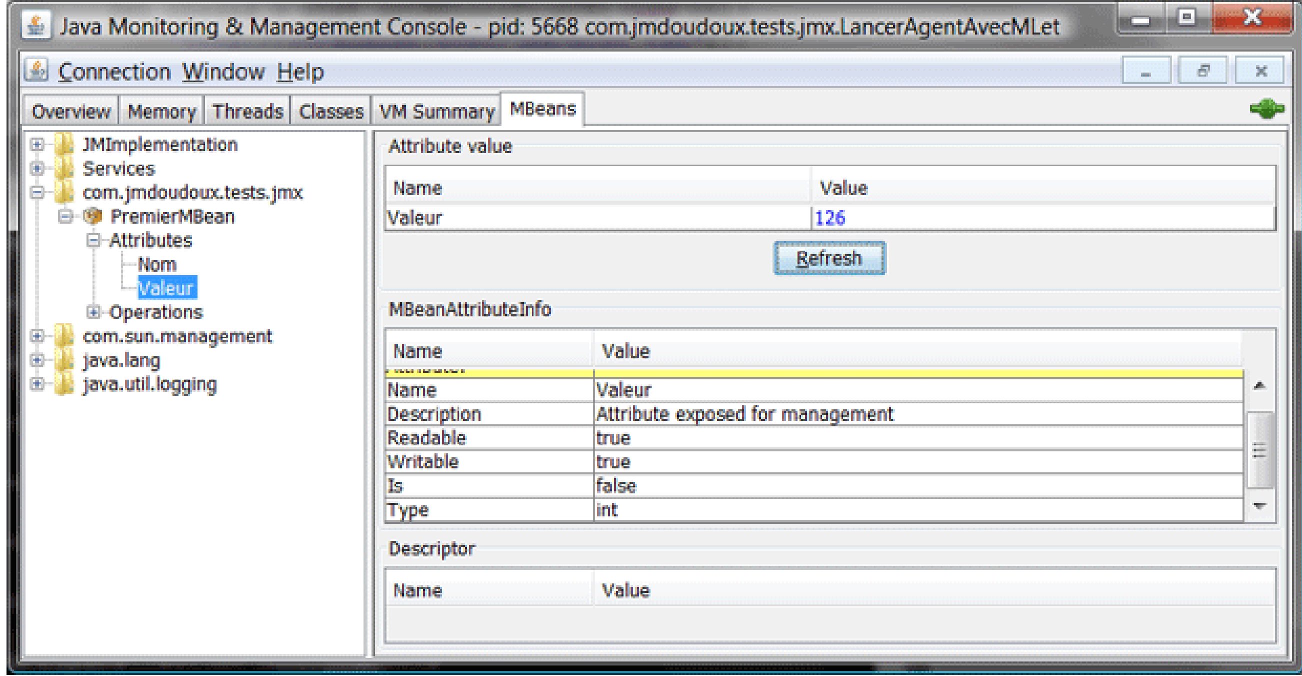
Dans l'exemple de cette section, c'est une simple webapp déployée dans un serveur Tomcat qui contient à sa racine les deux fichiers.

Il faut exécuter l'agent.

Résultat :

```
Instanciation et enregistrement du Service MLet

Lecture du fichier de configuration
Object = com.jmdoudoux.tests.jmx.Premier[com.jmdoudoux.tests.jmx:type=PremierMBean]

Classpath du service MLet : [http://localhost:8080/jmx/TestJMX.jar]

Recherche du mbean enregistré
ObjectName=com.jmdoudoux.tests.jmx:type=PremierMBean

Execution de l'agent ...
```

Il est possible de consulter le MBean enregistré dans l'agent par exemple avec JConsole.



31.7.2. Le service de type Timer

Le service Timer permet d'envoyer des notifications prédéfinies à un moment donné ou périodiquement. Ces notifications sont envoyées à tous les objets qui se sont abonnés pour recevoir les notifications émises par le service.

Les caractéristiques de l'émission d'une notification sont assez souple : elle démarre à une certaine date/heure et est répétée à intervalle de temps régulier durant une période ou pour un certain nombre d'occurrences.

Le service Timer est implémenté sous la forme d'un MBean ce qui permet de l'administrer au travers de JMX lui même.

Les notifications émises par le service Timer sont de type TimerNotification.

L'implémentation du service Timer est encapsulée dans la classe javax.management.timer.Timer

31.7.2.1. Les fonctionnalités du service Timer

Le service Timer est implémentée sous la forme d'un MBean. Pour l'utiliser, il faut l'instancier et l'enregistrer dans le serveur de MBeans.

Pour activer le service, il faut utiliser sa méthode start(). Pour le désactiver, il faut utiliser la méthode stop(). Avant l'appel à la méthode start() et après la méthode stop() aucune notification n'est émise même si les conditions d'une émission sont remplies. Si le service est redémarré et que la méthode setSendPastNotifications() a été invoquée avec le paramètre true alors les notifications ratées seront émises.

La méthode isActive() permet de savoir si le service est actif ou non.

Pour s'abonner aux notifications, un client ou une classe doivent s'enregistrer en tant que listener sur le MBean du service Timer. Chaque fois qu'une condition est remplie, une notification est envoyée à tous les abonnés.

Dès que les conditions d'une notification ne peuvent plus être remplies (par exemple si le nombre d'occurrences est atteint), la définition de la notification est supprimé automatiquement de la liste maintenue dans le service.

Il est possible de supprimer la définition d'une notification en utilisant la méthode removeNotification() qui attend l'identifiant de la définition de notifications. Il est aussi possible de supprimer plusieurs définitions en utilisant la méthode removeNotifications() qui attend en paramètre leur type : dans ce cas, elle va supprimer toutes les définitions de notifications qui ont le type fourni en paramètre.

La méthode removeAllNotifications() permet de supprimer toutes les notifications contenues dans le service.

31.7.2.2. L'ajout d'une définition de notifications

Le service Timer maintient une liste des définitions de notifications qu'il aura à traiter. La méthode addNotification() permet d'ajouter une définition de notifications. Cette méthode possède plusieurs surcharges qui possèdent toutes quatre paramètres en commun :

- type : chaîne de caractères qui précise le type de la notification
- message : chaîne de caractères qui contient le message de la notification
- userData : un objet qui encapsule des données dédiées à la notification
- date : date/heure de début d'émission des notifications

Plusieurs surcharges attendent en paramètres un ou plusieurs de paramètres ci dessous :

- period : intervalle de temps en millisecondes entre l'émission de deux notifications. La valeur 0 inhibe toute répétition.
- nbOccurences : nombre total d'émissions de notifications à réaliser. Avec la valeur 0 les émissions sont infinies.
- fireImmediate : booléen qui précise si l'émission de la première notification doit lieu immédiatement à l'ajout de la définition de notifications au service. La valeur true émet une notification immédiatement, la valeur false n'émet la première notification qu'un fois que les conditions de la définition sont remplies.

La méthode addNotification() peut lever une exception de type IllegalArgumentException si une ou plusieurs valeurs de ces paramètres est invalide, par exemple :

- date est null (Timer notification date cannot be null)
- period ou nbOccurences a une valeur négative (Negative values for the periodicity)

La méthode addNotification() renvoie un identifiant de la définition des notifications qui peut être utile notamment pour supprimer la définition.

Remarque : il n'est pas possible de modifier les paramètres d'une définition de notifications.

31.7.2.3. Un exemple de mise en oeuvre du service Timer

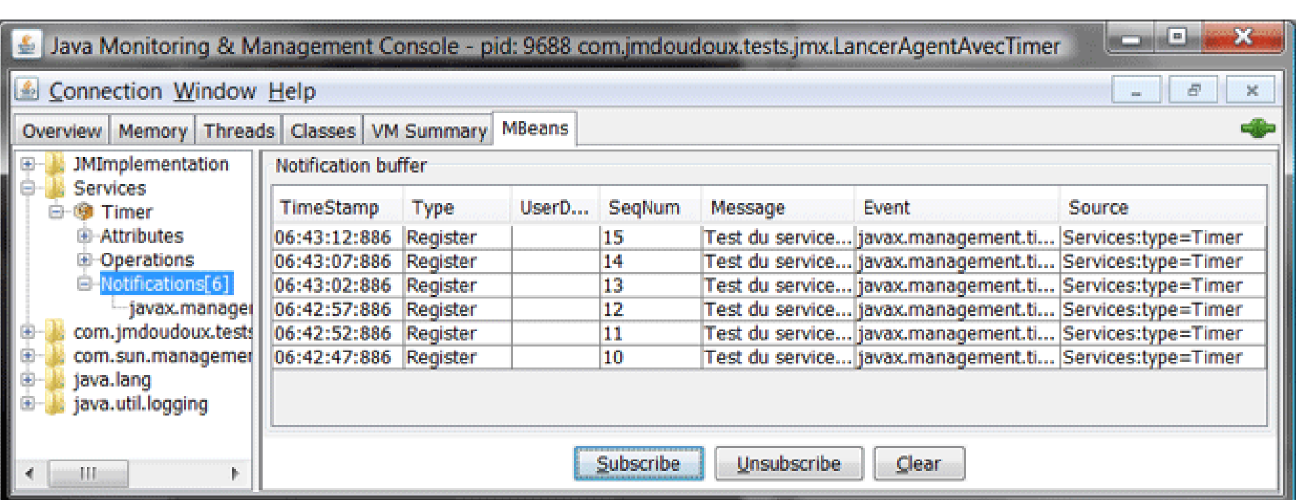
Dans l'exemple de cette section, une notification sera émise toutes les 5 secondes pour une durée infinie.

Exemple :

```
package com.jmdoudoux.tests.jmx;

import java.io.IOException;
import java.lang.management.ManagementFactory;
import java.net.MalformedURLException;
import java.util.Date;

import javax.management.InstanceAlreadyExistsException;
import javax.management.MBeanRegistrationException;
import javax.management.MBeanServer;
import javax.management.MalformedObjectNameException;
import javax.management.NotCompliantMBeanException;
import javax.management.ObjectName;
import javax.management.remote.JMXConnectorServer;
import javax.management.remote.JMXConnectorServerFactory;
import javax.management.remote.JMXServiceURL;

public class LancerAgentAvecTimer {

  public static void main(String[] args) {
    MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();

    try {
```

```
      // instanciation et enregistrement du service Timer
      javax.management.timer.Timer timer = new javax.management.timer.Timer();
      mbs.registerMBean(timer, new ObjectName("Services:type=Timer"));

      // ajout de la définition des notifications
      timer.addNotification("Register", "Test du service timer", new String(),
          new Date(), 5000, 0);
      timer.setSendPastNotifications(false);
      timer.start();

      // Creation et démarrage du connecteur RMI
      JMXServiceURL url = new JMXServiceURL(
          "service:jmx:rmi:///jndi/rmi://localhost:9000/server");
      JMXConnectorServer cs = JMXConnectorServerFactory.newJMXConnectorServer(
          url, null, mbs);
      cs.start();
      System.out.println("Lancement connecteur RMI " + url);

      while (true) {
        Thread.sleep(1000);
      }

    } catch (MalformedObjectNameException e) {
      e.printStackTrace();
    } catch (NullPointerException e) {
      e.printStackTrace();
    } catch (InstanceAlreadyExistsException e) {
      e.printStackTrace();
    } catch (MBeanRegistrationException e) {
      e.printStackTrace();
    } catch (NotCompliantMBeanException e) {
      e.printStackTrace();
    } catch (InterruptedException e) {
      e.printStackTrace();
    } catch (MalformedURLException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Pour voir les notifications dans JConsole, il faut sélectionner « Notitifications » pour le MBean du service Timer et cliquer sur le bouton « Subscribe ».



31.7.3. Le service de type Monitor

La suite de cette section sera développée dans une version future de ce document

31.7.4. Le service de type Relation

La suite de cette section sera développée dans une version future de ce document

31.8. La couche services distribués

Il existe de nombreuses applications de gestion et de monitoring qui utilisent des protocoles standards comme SNMP (Simple Network Management Protocol) ou propriétaires.

Une application de gestion permet à des utilisateurs d'interagir avec les MBeans en communiquant avec le serveur de MBeans ou un service de l'agent JMX. Une application web utilisant un adaptateur de protocoles pour HTML ou une application utilisant un adaptateur de protocoles pour SNMP sont des exemples d'applications de gestion. Ces applications peuvent ne pas implémenter l'API JMX et dans ce cas elle communique avec l'agent JMX via un protocole dédié.

La couche services distribuées fournies des interfaces et des composants pour permettre à des outils distants de communiquer avec l'agent JMX.

La couche services distribuées contient une application de gestion qui va interagir avec l'agent JMX. Les clients JMX distants s'exécutent dans une JVM différente de celle du serveur de MBeans et utilise un protocole pour communiquer avec ce dernier.

Un client JMX distant ne peut pas utiliser le serveur de MBeans directement. Les composants de cette couche permettent d'accéder à un serveur de MBeans au travers de différents protocoles.

La manière dont un agent JMX est accédé au travers du réseaux est spécifié par la JSR 160 (JMX Remoting) qui est une fonctionnalité de la version 1.2 de JMX.

31.8.1. L'interface MBeanServerConnection

Un client JMX distant manipulent des MBeans en utilisant un objet qui implémente l'interface MBeanServerConnection. Une instance de l'interface MBeanServerConnection permet de se connecter à un serveur de MBeans local ou distant et d'interagir avec lui.

Pour utiliser un MBean local, il est possible d'utiliser directement le serveur de MBeans. Pour accéder à un MBean distant, il faut obligatoirement utiliser une instance de l'interface MBeanServerConnection.

Le client utilise les méthodes de l'interface MBeanServerConnection pour accéder aux fonctionnalités exposées par un MBean :

- createMBean() : pour instancier et enregister un MBean dans le serveur de MBeans
- getAttribute() : pour obtenir la valeur d'un attribut d'un MBean
- setAttribute() : pour mettre à jour la valeur d'un attribut d'un MBean
- invoke() : pour invoquer un constructeur ou une méthode d'un MBean
- isRegistred() : pour déterminer si un MBean est déjà enregistré dans le serveur de MBeans
- queryMBeans() : pour obtenir une collection des MBeans enregistrés dans le serveur de MBeans
- queryNames() : pour obtenir une collection des noms des MBeans enregistrés dans le serveur de MBeans

31.8.2. Les connecteurs et les adaptateurs de protocoles

Pour permettre la communication entre un agent et un client JMX, JMX propose des adaptateurs de protocole ou des connecteurs qui se chargent de la communication entre l'application de gestion et l'agent JMX avec un protocole particulier.

La communication entre un agent JMX et une application de gestion peut donc être assurée par deux mécanismes :

- les connecteurs (connectors) : ils assurent la communication entre l'application de gestion et l'agent JMX. Le protocole utilisé par défaut est RMI. JMX définit aussi un protocole reposant sur des sockets TCP dont le support est optionnel nommé JMX Messaging Protocol (JMXMP). Un connecteur permet à une application distante de communiquer avec l'agent en utilisant un connecteur côté client et côté serveur respectant les spécifications de JMX.
- les adaptateurs de protocoles (protocol adaptors) : ils permettent à une application de manipuler les MBeans enregistrés dans un serveur de MBeans en utilisant un certain protocole (par exemple SNMP ou l'adaptateur de protocole pour HTML qui permet de manipuler les MBeans d'un agent JMX au moyen d'un simple navigateur web). Les interactions avec les MBeans se font au travers de ce protocole : les actions à réaliser sont convertit de et vers le protocole utilisé.

Les connecteurs et les adaptateurs de protocoles permettent l'accès aux MBeans enregistrés dans un agent JMX à une application distant. Ils permettent un accès aux services de l'agent et aux MBeans enregistrés dans celui ci.

Ainsi pour être utilisable, un agent JMX doit fournir ou moins un connecteur ou un adaptateur de protocoles. La plate-forme Java SE fournit en standard un connecteur reposant sur RMI.

Un agent peut être accédé via plusieurs connecteurs ou adaptateurs de protocoles simultanément.

31.8.2.1. Les connecteurs

Un connecteur permet le dialogue entre l'agent et l'application de gestion distante via un protocole dédié. Un connecteur est composé d'une partie cliente liée à l'application de gestion et d'une partie serveur liée à l'agent JMX. La partie serveur du connecteur attends les connexions de la partie cliente : c'est donc la partie cliente qui est responsable de l'initialisation de la connexion.

Un connecteur permet donc à un client JMX distant de communiquer avec un agent JMX. L'agent JMX est chargé d'initialiser et de configurer le connecteur côté serveur. Le client JMX est chargé d'initialiser et de configurer le connecteur côté client.

Un connecteur permet d'obtenir une instance de l'interface MBeanServerConnection.

Les spécifications de l'API JMX Remote définies trois catégories de connecteurs :

- connecteur RMI : ce type de connecteur utilise la technologie RMI de Java. L'implémentation de ce type de connecteur est obligatoire.

* connecteur générique : ce type de connecteur utilise des sockets TCP et le protocole JMXMP (JMX Messaging Protocol). L'implémentation de ce type de connecteur est optionnel.
* connecteur utilisant protocole spécifique : ce type de connecteur utilise un protocole qui n'est pas défini par JMX

Le connecteur RMI peut utiliser les deux standards de transport de RMI :

* Java Remote Method Protocol (JRMP)
* Internet Inter-ORB Protocol (IIOP)

31.8.2.2. Les adaptateurs de protocoles

Les adaptateurs de protocoles permettent un accès à un agent JMX au travers d'un protocole particulier. Celui ci peut par exemple être un protocole propriétaire utilisé par une application de gestion commerciale.

Un adaptateur de protocoles permet à une application qui n'utilise pas JMX de communiquer avec un agent JMX sans que la partie cliente n'utilise aucune API de JMX.

Par exemple, l'adaptateur HTML fourni avec l'implémentation de référence de JMX permet d'interagir avec les MBeans d'un agent JMX utilisant cet adaptateur avec un simple navigateur web.

31.8.3. L'utilisation du connecteur RMI

Pour utiliser le connecteur RMI, l'agent JMX doit créer un connecteur RMI et le lancer.

Le connecteur est accessible via une url encapsulée dans la classe JMXServiceURL.

La partie serveur d'un connecteur est encapsulée dans la classe JMXConnectorServer.

Une instance de la classe JMXConnectorServer est obtenue en utilisant la méthode newJMXConnectorServer() de la fabrique JMXConnectorServerFactory.

Exemple :

```
package com.jmdoudoux.tests.jmx;

import java.io.IOException;
import java.lang.management.ManagementFactory;
import java.net.MalformedURLException;

import javax.management.InstanceAlreadyExistsException;
import javax.management.MBeanRegistrationException;
import javax.management.MBeanServer;
import javax.management.MalformedObjectNameException;
import javax.management.NotCompliantMBeanException;
import javax.management.ObjectName;
import javax.management.remote.JMXConnectorServer;
import javax.management.remote.JMXConnectorServerFactory;
import javax.management.remote.JMXServiceURL;

public class LancerAgentAvecRMI {

  public static void main(String[] args) {

    System.out.println("Lancement de l'agent JMX");

    MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();

    ObjectName name = null;
    try {
```

```
      System.out.println("Instanciation et enregistrement du MBean");

      name = new ObjectName("com.jmdoudoux.tests.jmx:type=PremierMBean");

      Premier mbean = new Premier();

      mbs.registerMBean(mbean, name);

      // Creation et demarrage du connecteur RMI
      JMXServiceURL url = new JMXServiceURL(
          "service:jmx:rmi:///jndi/rmi://localhost:9000/server");
      JMXConnectorServer cs = JMXConnectorServerFactory.newJMXConnectorServer(
          url, null, mbs);
      cs.start();
      System.out.println("Lancement connecteur RMI "+url);

      int i = 0;
      System.out.println("Incrementation de la valeur du MBean ...");
      while (i < 60) {

        mbean.setValeur(mbean.getValeur() + 1);
        Thread.sleep(1000);
        i++;
      }

      System.out.println("Arret connecteur RMI ");
      cs.stop();

      System.out.println("Arret de l'agent JMX");

    } catch (MalformedObjectNameException e) {
      e.printStackTrace();
    } catch (NullPointerException e) {
      e.printStackTrace();
    } catch (InstanceAlreadyExistsException e) {
      e.printStackTrace();
    } catch (MBeanRegistrationException e) {
      e.printStackTrace();
    } catch (NotCompliantMBeanException e) {
      e.printStackTrace();
    } catch (InterruptedException e) {
      e.printStackTrace();
    } catch (MalformedURLException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

L'appel à la méthode close() est important car il permet notamment de supprimer le connecteur enregistré dans le registre RMI.

Un exception de type javax.naming.NameAlreadyBoundException est levée si l'agent est lancé et que le connecteur est déjà enregistré dans le registre RMI. Dans ce cas, il faut redémarrer le registre.

Exemple :

```
C:\Users\Jean Michel\workspace\TestJMX\bin>java com.jmdoudoux.tests.jmx.LancerAg
entAvecRMI
Lancement de l'agent JMX
Instanciation et enregistrement du MBean
Lancement connecteur RMI service:jmx:rmi:///jndi/rmi://localhost:9000/server
Incrementation de la valeur du MBean ...

C:\Users\Jean Michel\workspace\TestJMX\bin>java com.jmdoudoux.tests.jmx.LancerAg
entAvecRMI
Lancement de l'agent JMX
Instanciation et enregistrement du MBean
java.io.IOException: Cannot bind to URL [rmi://localhost:9000/server]: javax.nam
ing.NameAlreadyBoundException: server [Root exception is java.rmi.AlreadyBoundEx
```

```
ception: server]
        at javax.management.remote.rmi.RMIConnectorServer.newIOException(RMIConn
ectorServer.java:804)
        at javax.management.remote.rmi.RMIConnectorServer.start(RMIConnectorServ
er.java:417)
        at com.jmdoudoux.tests.jmx.LancerAgentAvecRMI.main(LancerAgentAvecRMI.ja
va:40)
Caused by: javax.naming.NameAlreadyBoundException: server [Root exception is jav
a.rmi.AlreadyBoundException: server]
        at com.sun.jndi.rmi.registry.RegistryContext.bind(RegistryContext.java:1
22)
        at com.sun.jndi.toolkit.url.GenericURLContext.bind(GenericURLContext.jav
a:208)
        at javax.naming.InitialContext.bind(InitialContext.java:400)
        at javax.management.remote.rmi.RMIConnectorServer.bind(RMIConnectorServe
```

Pour utiliser un connecteur RMI, il faut obligatoirement lancer un registre RMI

Exemple :

```
C:\>rmiregistry 9000
```

Le paramètre précisé à la commande rmiregistry est le port utilisé pour les communications.

Le client JMX doit obtenir une instance du type MBeanServerConnection qui va permettre de se connecter sur le serveur de MBeans de l'agent JMX. Une telle instance est obtenue en utilisant un objet de type JMXConnector fournie par l'invocation de la méthode connect() de la fabrique JMXConnectorFactory. La méthode connect() attend en premier paramètre l'url de connexion sur le serveur de MBeans de l'agent JMX.

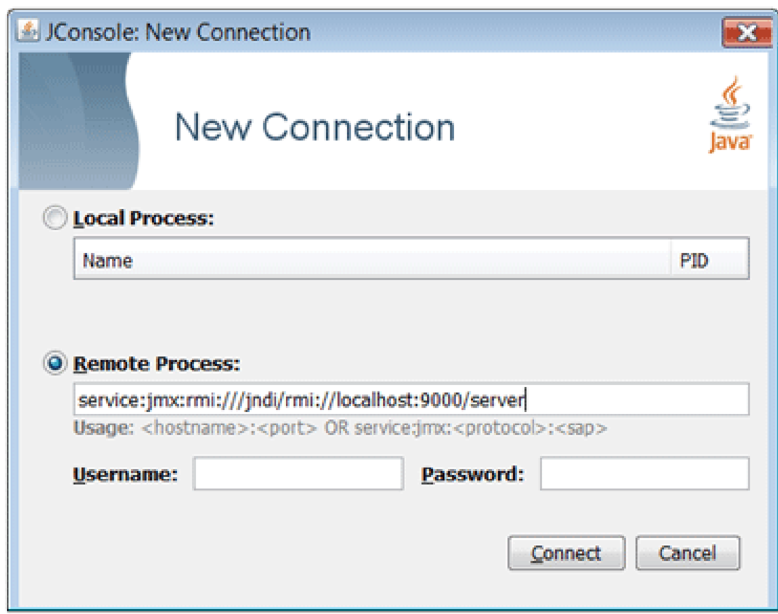
La méthode getMBeanServerConnection() de l'instance de type JMXConnector renvoie un objet de type MBeanServerConnection qui permet d'interagir avec le serveur de MBeans.

Exemple :

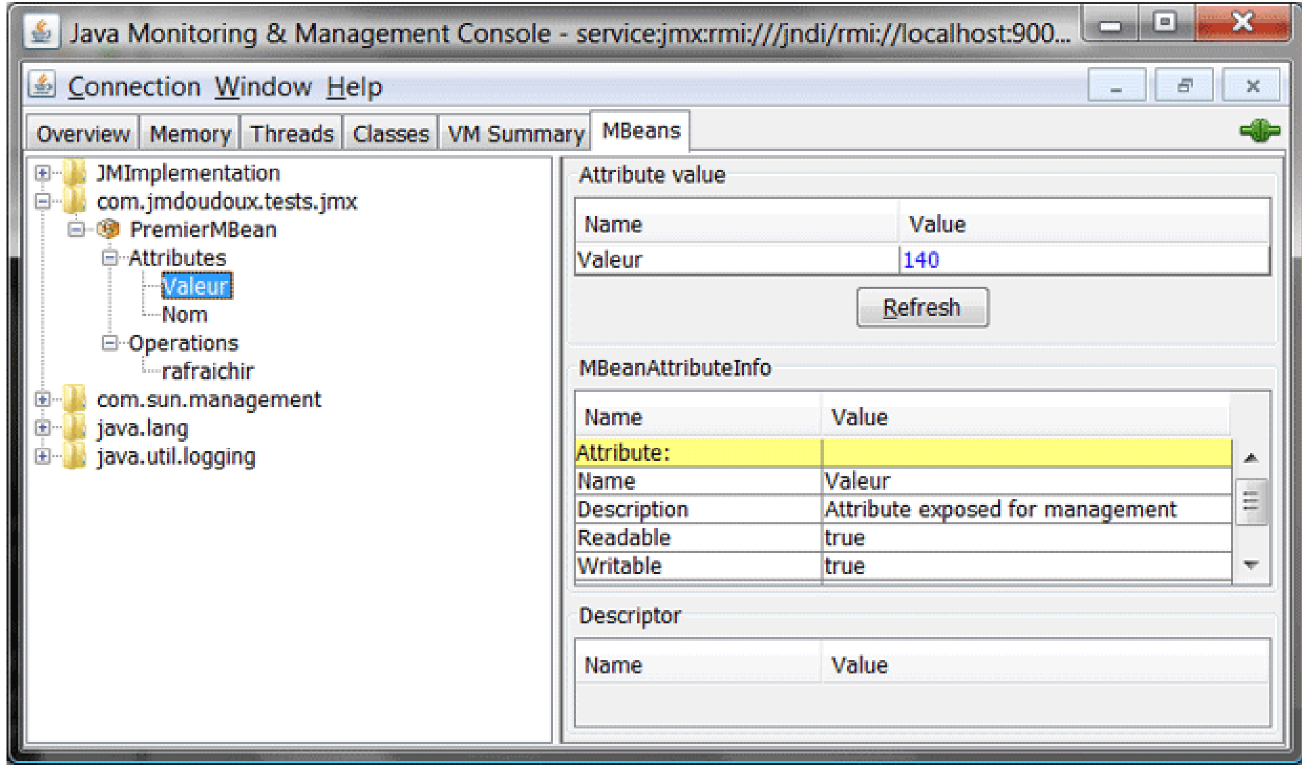
```
package com.jmdoudoux.tests.jmx;

import java.io.IOException;
import java.net.MalformedURLException;

import javax.management.MBeanServerConnection;
import javax.management.MBeanServerInvocationHandler;
import javax.management.MalformedObjectNameException;
import javax.management.ObjectName;
import javax.management.remote.JMXConnector;
import javax.management.remote.JMXConnectorFactory;
import javax.management.remote.JMXServiceURL;

public class ClientJMXAvecRMI {
  public static void main(String[] args) {
    MBeanServerConnection mbsc = null;
    JMXConnector connecteur = null;

    ObjectName name = null;
    try {
      name = new ObjectName("com.jmdoudoux.tests.jmx:type=PremierMBean");

      JMXServiceURL url = new JMXServiceURL(
          "service:jmx:rmi:///jndi/rmi://localhost:9000/server");

      connecteur = JMXConnectorFactory.connect(url, null);

      mbsc = connecteur.getMBeanServerConnection();

      PremierMBean mbean = (PremierMBean) MBeanServerInvocationHandler
          .newProxyInstance(mbsc, name, PremierMBean.class, false);
      int valeur = mbean.getValeur();
      System.out.println("valeur = " + valeur);
```

```
    } catch (MalformedObjectNameException e) {
      e.printStackTrace();
    } catch (NullPointerException e) {
      e.printStackTrace();
    } catch (MalformedURLException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    } finally {
      if (mbsc != null) {
        try {
          connecteur.close();
        } catch (IOException e) {
          e.printStackTrace();
        }
      }
    }

  }
}
```

Il est aussi possible d'utiliser un autre client JMX, par exemple l'outil JConsole



L'onglet MBean permet de retrouver le MBean enregistré dans le serveur de MBeans et d'interagir avec lui.



31.8.4. L'utilisation du connecteur utilisant le protocole JMXMP

Un connecteur générique utilise le protocole JMXMP qui repose sur des sockets avec le protocole TCP pour les communications, la sérialisation pour l'échange des objets et les API standards de Java pour la sécurité notamment JSSE et JASS.

L'utilisation du connecteur JMXMP est simple. Pour utiliser ce protocole, il faut une implémentation de ce protocole par exemple celle fournie avec l'implémentation de référence de la JSR 160. Il suffit alors d'ajouter la bibliothèque jmxremote_optional.jar dans le classpath.

Le protocole JMXMP permet d'échanger des objets Java sérialisés via une connexion TCP. La communication entre le client et le serveur de MBeans n'a lors besoin que de définir un port de communication.

Le code de l'agent est similaire à celui de l'agent utilisant le connecteur RMI avec cependant une url de connexion dédiée au protocole JMXMP

Exemple :

```
package com.jmdoudoux.tests.jmx;

import java.io.IOException;
import java.lang.management.ManagementFactory;
import java.net.MalformedURLException;

import javax.management.InstanceAlreadyExistsException;
import javax.management.MBeanRegistrationException;
import javax.management.MBeanServer;
import javax.management.MalformedObjectNameException;
import javax.management.NotCompliantMBeanException;
import javax.management.ObjectName;
import javax.management.remote.JMXConnectorServer;
import javax.management.remote.JMXConnectorServerFactory;
import javax.management.remote.JMXServiceURL;

public class LancerAgentAvecJMXMP {

  public static void main(String[] args) {
```

```
    System.out.println("Lancement de l'agent JMX");

    MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();

    ObjectName name = null;
    try {
      System.out.println("Instanciation et enregistrement du Mbean");

      name = new ObjectName("com.jmdoudoux.tests.jmx:type=PremierMBean");

      Premier mbean = new Premier();

      mbs.registerMBean(mbean, name);

      // Creation et demarrage du connecteur pour le protocole JMXMP
      JMXServiceURL url = new JMXServiceURL(
          "service:jmx:jmxmp://localhost:9998");

      JMXConnectorServer cs = JMXConnectorServerFactory.newJMXConnectorServer(url, null, mbs);
      cs.start();

      System.out.println("Lancement connecteur pour le protocle JMXMP " + url);

      int i = 0;
      System.out.println("Incrementation de la valeur du MBean ...");
      while (i < 6000) {

        mbean.setValeur(mbean.getValeur() + 1);
        Thread.sleep(1000);
        i++;
      }

      System.out.println("Arret connecteur pour le protocle JMXMP ");
      cs.stop();

      System.out.println("Arret de l'agent JMX");

    } catch (MalformedObjectNameException e) {
      e.printStackTrace();
    } catch (NullPointerException e) {
      e.printStackTrace();
    } catch (InstanceAlreadyExistsException e) {
      e.printStackTrace();
    } catch (MBeanRegistrationException e) {
      e.printStackTrace();
    } catch (NotCompliantMBeanException e) {
      e.printStackTrace();
    } catch (InterruptedException e) {
      e.printStackTrace();
    } catch (MalformedURLException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Le code du client est aussi similaire à celui du client utilisant le connecteur RMI avec l'utilisation de l'url dédiée.

Exemple :

```
package com.jmdoudoux.tests.jmx;

import java.io.IOException;
import java.net.MalformedURLException;

import javax.management.MBeanServerConnection;
import javax.management.MBeanServerInvocationHandler;
import javax.management.MalformedObjectNameException;
import javax.management.ObjectName;
import javax.management.remote.JMXConnector;
```

```
import javax.management.remote.JMXConnectorFactory;
import javax.management.remote.JMXServiceURL;

public class ClientJMXAvecJMXMP {
  public static void main(String[] args) {
    MBeanServerConnection mbsc = null;
    JMXConnector connecteur = null;

    ObjectName name = null;
    try {
      name = new ObjectName("com.jmdoudoux.tests.jmx:type=PremierMBean");

      JMXServiceURL url = new JMXServiceURL(
          "service:jmx:jmxmp://localhost:9998");

      connecteur = JMXConnectorFactory.connect(url, null);

      mbsc = connecteur.getMBeanServerConnection();

      PremierMBean mbean = (PremierMBean) MBeanServerInvocationHandler
          .newProxyInstance(mbsc, name, PremierMBean.class, false);
      int valeur = mbean.getValeur();
      System.out.println("valeur = " + valeur);
      mbean.rafraichir();

    } catch (MalformedObjectNameException e) {
      e.printStackTrace();
    } catch (NullPointerException e) {
      e.printStackTrace();
    } catch (MalformedURLException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    } finally {
      if (mbsc != null) {
        try {
          connecteur.close();
        } catch (IOException e) {
          e.printStackTrace();
        }
      }
    }

  }
}
```

L'exécution du client sans mettre la bibliothèque jmxremote_optional.jar lève une exception de type MalformedURLException

Résultat :

```
C:\Users\Jean Michel\workspace\TestJMX\bin>java -cp . com.jmdoudoux.tests.jmx.La
ncerAgentAvecJMXMP
Lancement de l'agent JMX
Instanciation et enregistrement du Mbean
java.net.MalformedURLException: Unsupported protocol: jmxmp
        at javax.management.remote.JMXConnectorServerFactory.newJMXConnectorServ
er(JMXConnectorServerFactory.java:323)
        at com.jmdoudoux.tests.jmx.LancerAgentAvecJMXMP.main(LancerAgentAvecJMXM
P.java:39)
```

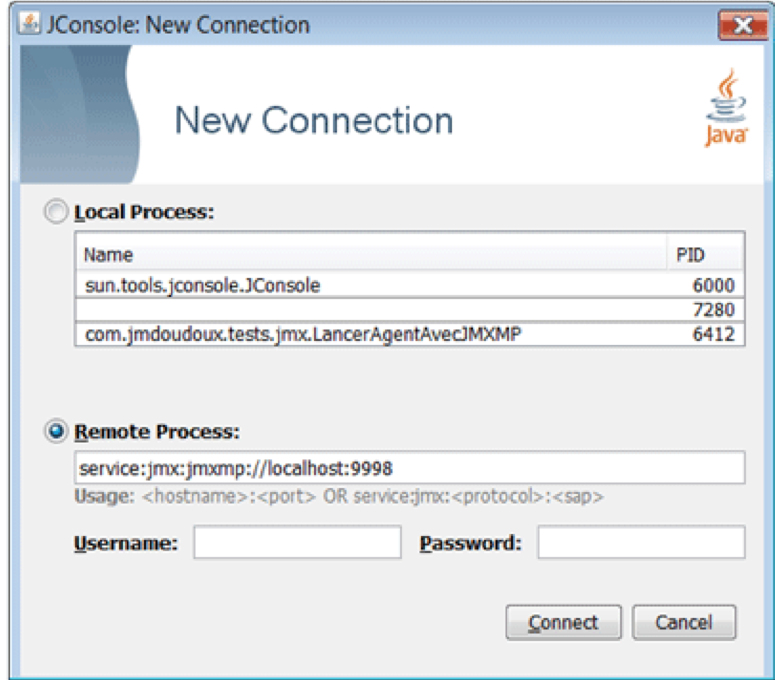
Avec la bibliothèque ajoutée au classpath, le client peut se connecter à l'agent et interagir avec le MBean.

Exemple :

```
C:\Users\Jean Michel\workspace\TestJMX\bin>java -cp
.;C:\java\api\jmxremote-1_0_
1-bin\lib\jmxremote_optional.jar
com.jmdoudoux.tests.jmx.ClientJMXAvecJMXMP
```

```
valeur = 867
```

Il est aussi d'utiliser JConsole pour se connecter à l'agent en utilisant comme paramètre de connexion l'url utilisée par l'agent.



Dans ce cas pour que la connexion réussisse, il faut ajouter la bibliothèque au classpath pour permettre à JConsole d'avoir l'implémentation du protocole JMXMP. Le plus simple est d'ajouter le fichier jar dans le sous répertoire lib/ext du répertoire d'installation du JRE.

31.8.5. L'utilisation de l'adaptateur de protocole HTML

L'adaptateur de protocoles HTML permet d'accéder à un agent en utilisant le protocole HTML : ainsi un simple navigateur web peut faire office de client JMX.

L'adaptateur de protocoles HTML est implémenté sous la forme d'un MBean et peut donc à ce titre être géré comme tout MBean.

Bien que JMX soit intégré à Java 5, l'adaptateur de protocole HTML n'est pas fourni en standard avec le JDK. Il faut télécharger l'implémentation de référence de JMX à l'url :

http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement/download.jsp

L'archive jmx-1_2_1-ri.zip contient dans le sous répertoire lib une bibliothèque nommée jmxtools.jar qui doit être ajoutée au classpath.

L'adaptateur est encapsulé dans la classe com.sun.jdmk.comm.HtmlAdaptorServer : c'est un MBean qui doit être instancié et enregistré dans le serveur de MBeans de l'agent.

Le port utilisé par l'adaptateur doit être précisé en utilisant la méthode setPort(). Par défaut, c'est le port 8082 qui est utilisé.

La méthode start() permet de démarrer l'adaptateur et lui permettre de traiter les requêtes HTML.

Exemple :

```
package com.jmdoudoux.tests.jmx;
```

```
import java.lang.management.ManagementFactory;

import javax.management.InstanceAlreadyExistsException;
import javax.management.MBeanRegistrationException;
import javax.management.MBeanServer;
import javax.management.MalformedObjectNameException;
import javax.management.NotCompliantMBeanException;
import javax.management.ObjectName;

import com.sun.jdmk.comm.HtmlAdaptorServer;

public class LancerAgentAvecHTMLAdaptateur {
  static final int PORT_ADAPTATEUR = 8000;

  public static void main(String[] args) {

    System.out.println("Lancement de l'agent JMX");

    MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();

    ObjectName name = null;
    ObjectName adapterName = null;

    try {
      System.out.println("Instanciation et enregistrement du MBean");

      name = new ObjectName("com.jmdoudoux.tests.jmx:type=PremierMBean");

      Premier mbean = new Premier();

      mbs.registerMBean(mbean, name);

      // Creation et demarrage de l'adaptateur de protocole HTML
      HtmlAdaptorServer adapter = new HtmlAdaptorServer();
      adapterName = new ObjectName(
          "com.jmdoudoux.tests.jmx:name=htmladaptor,port=" + PORT_ADAPTATEUR);
      adapter.setPort(PORT_ADAPTATEUR);
      mbs.registerMBean(adapter, adapterName);
      adapter.start();
      System.out
          .println("Lancement de l'adaptateur de protocole HTML sur le port "
              + PORT_ADAPTATEUR);

      int i = 0;
      System.out.println("Incrementation de la valeur du MBean ...");
      while (i < 600) {

        mbean.setValeur(mbean.getValeur() + 1);
        Thread.sleep(1000);
        i++;
      }

      System.out.println("Arret de l'adaptateur de protocole HTML ");
      adapter.stop();

      System.out.println("Arret de l'agent JMX");

    } catch (MalformedObjectNameException e) {
      e.printStackTrace();
    } catch (NullPointerException e) {
      e.printStackTrace();
    } catch (InstanceAlreadyExistsException e) {
      e.printStackTrace();
    } catch (MBeanRegistrationException e) {
      e.printStackTrace();
    } catch (NotCompliantMBeanException e) {
      e.printStackTrace();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```

Après compilation des classes, il faut exécuter l'agent JMX

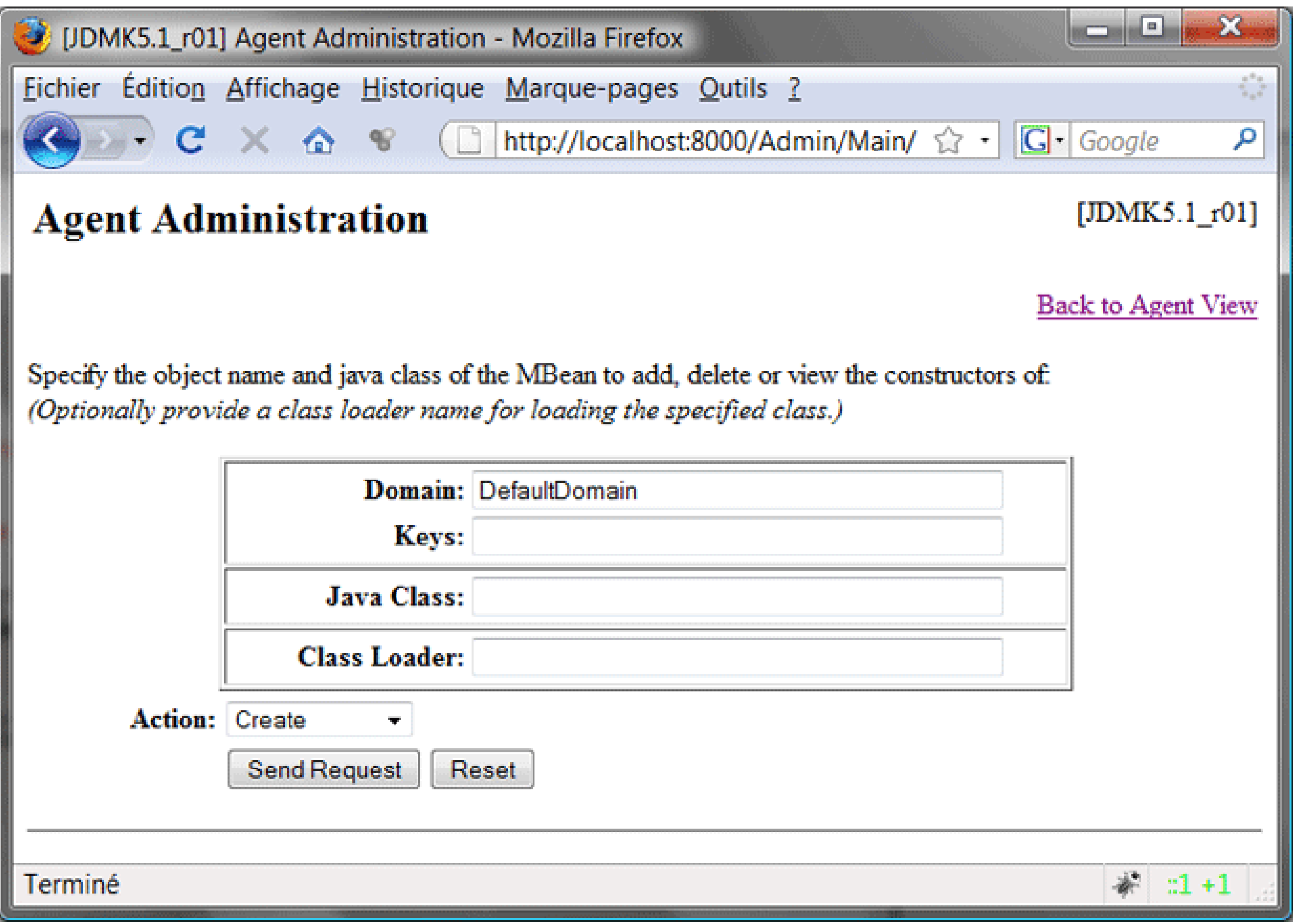
Exemple :

```
C:\Users\Jean Michel\workspace\TestJMX\bin>java -cp .;jmxtools.jar com.jmdoudoux
.tests.jmx/LancerAgentAvecHTMLAdaptateur
Lancement de l'agent JMX
Instanciation et enregistrement du MBean
Lancement de l'adaptateur de protocole HTML sur le port 8000
Incrementation de la valeur du MBean ...
Arret de l'adaptateur de protocole HTML
Arret de l'agent JMX
```

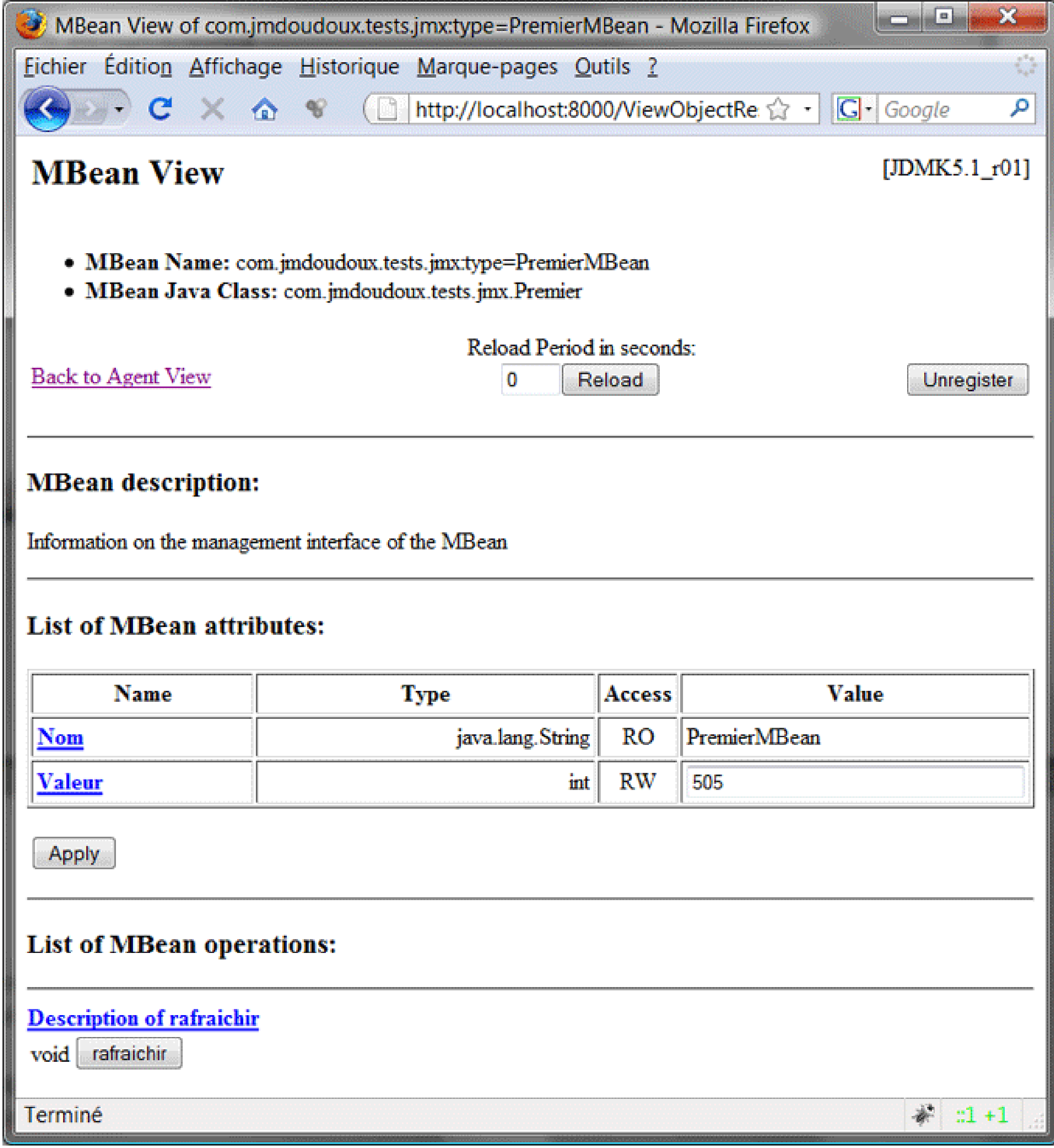
Il faut ouvrir un navigateur avec l'url http://localhost:8000.

La première page affichée par l'adaptateur affiche les MBeans enregistrés pour le domaine par défaut dans le serveur de MBeans de l'agent JMX.

Le bouton Admin affiche une page qui permet d'enregistrer un nouvel MBean dans le serveur.



L'adaptateur lui même est un MBean ce qui explique qu'il apparaît avec le MBean PremierMBean. En cliquant sur le lien de ce dernier, l'adaptateur affiche une page avec les propriétés et les opérations du MBean



Cette page affiche les propriétés, permet de modifier celles qui possèdent un setter et permet l'invocation des méthodes définies dans l'interface du MBean.

31.8.6. L'invocation d'un MBean via un proxy

Le classe MBeanServerConnection propose plusieurs méthodes pour interagir avec un MBean notamment :

- Object getAttribut() pour obtenir la valeur d'un attribut
- void setAttribut(ObjectName, Attribute) pour modifier la valeur d'un attribut
- Object invoke() pour invoquer une opération

Exemple :

```
package com.jmdoudoux.tests.jmx;
```

```
import java.io.IOException;
import java.net.MalformedURLException;

import javax.management.AttributeNotFoundException;
import javax.management.InstanceNotFoundException;
import javax.management.MBeanException;
import javax.management.MBeanServerConnection;
import javax.management.MalformedObjectNameException;
import javax.management.ObjectName;
import javax.management.ReflectionException;
import javax.management.remote.JMXConnector;
import javax.management.remote.JMXConnectorFactory;
import javax.management.remote.JMXServiceURL;

public class ClientJMXAvecRMI {
  public static void main(String[] args) {
    MBeanServerConnection mbsc = null;
    JMXConnector connecteur = null;

    ObjectName name = null;
    try {
      name = new ObjectName("com.jmdoudoux.tests.jmx:type=PremierMBean");

      JMXServiceURL url = new JMXServiceURL(
          "service:jmx:rmi:///jndi/rmi://localhost:9000/server");

      connecteur = JMXConnectorFactory.connect(url, null);

      mbsc = connecteur.getMBeanServerConnection();

      System.out.println("valeur = " + mbsc.getAttribute(name, "Valeur"));
      mbsc.invoke(name, "rafraichir", null, null);

    } catch (MalformedObjectNameException e) {
      e.printStackTrace();
    } catch (NullPointerException e) {
      e.printStackTrace();
    } catch (MalformedURLException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    } catch (AttributeNotFoundException e) {
      e.printStackTrace();
    } catch (InstanceNotFoundException e) {
      e.printStackTrace();
    } catch (MBeanException e) {
      e.printStackTrace();
    } catch (ReflectionException e) {
      e.printStackTrace();
    } finally {
      if (mbsc != null) {
        try {
          connecteur.close();
        } catch (IOException e) {
          e.printStackTrace();
        }
      }
    }

  }
}
```

L'utilisation de ces méthodes peut être source de soucis surtout car certains problèmes ne peuvent être détectés qu'à l'exécution par exemple si le nom fourni de la méthode à invoquer est erroné.

La classe MBeanServerInvocationHandler permet de créer un proxy qui va permettre d'interagir avec un MBean du serveur de MBeans. Cette classe va utiliser l'interface du MBean pour générer dynamiquement une classe de type proxy permettant d'interagir avec le MBean.

La méthode statique newInstanceProxy() permet de créer un proxy pour invoquer un MBean. Elle attend en paramètre

l'instance qui encapsule la connexion vers le serveur de MBeans, le nom identifiant le MBean, le type de l'interface du MBean, et un booléen qui permet de préciser si le proxy doit implémenter l'interface NotificationEmitter permettant d'abonner un listener aux notifications du MBean.

L'exemple ci dessous est identique à l'exemple précédent.

Exemple :

```
package com.jmdoudoux.tests.jmx;

import java.io.IOException;
import java.net.MalformedURLException;

import javax.management.MBeanServerConnection;
import javax.management.MBeanServerInvocationHandler;
import javax.management.MalformedObjectNameException;
import javax.management.ObjectName;
import javax.management.remote.JMXConnector;
import javax.management.remote.JMXConnectorFactory;
import javax.management.remote.JMXServiceURL;

public class ClientJMXAvecRMI {
  public static void main(String[] args) {
    MBeanServerConnection mbsc = null;
    JMXConnector connecteur = null;

    ObjectName name = null;
    try {
      name = new ObjectName("com.jmdoudoux.tests.jmx:type=PremierMBean");

      JMXServiceURL url = new JMXServiceURL(
          "service:jmx:rmi:///jndi/rmi://localhost:9000/server");

      connecteur = JMXConnectorFactory.connect(url, null);

      mbsc = connecteur.getMBeanServerConnection();

      PremierMBean mbean = (PremierMBean) MBeanServerInvocationHandler
          .newProxyInstance(mbsc, name, PremierMBean.class, false);
      int valeur = mbean.getValeur();
      System.out.println("valeur = " + valeur);
      mbean.rafraichir();

    } catch (MalformedObjectNameException e) {
      e.printStackTrace();
    } catch (NullPointerException e) {
      e.printStackTrace();
    } catch (MalformedURLException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    } finally {
      if (mbsc != null) {
        try {
          connecteur.close();
        } catch (IOException e) {
          e.printStackTrace();
        }
      }
    }

  }
}
```

Le code de la création du proxy est un peu particulier mais son utilisation facilite l'écriture du code qui invoque les fonctionnalités du MBean puisque le proxy rend les appels distants comme si ces appels étaient locaux. Le code est ainsi plus facile à écrire et à lire.

31.8.7. La recherche et la découverte des agents JMX

L'API JMX Remote précise comment il est possible de rechercher et découvrir des agents JMX en utilisant des infrastructures et des API existantes (aucune nouvelle API n'est définie par les spécifications JMX).

Un client JMX se connecte à un agent JMX via un connecteur ou un adaptateur de protocoles. Pour rechercher et découvrir un agent, un agent peut utiliser trois possibilités optionnelles d'infrastructures :

- Service Location Protocol (SLP)
- la technologie Jini
- JNDI avec un annuaire

31.8.7.1. Via le Service Location Protocol (SLP)

Le Service Location Protocol (SLP) est un framework qui permet de rechercher, découvrir et connaître la configuration de services au travers du réseau.

L'agent JMX doit enregistrer chacun de ces connecteurs auprès du registre de SLP en lui fournissant son adresse et des attributs obligatoires et éventuellement optionnels pour qualifier le connecteur.

Le client JMX interroge le registre de SLP pour obtenir les adresses éventuellement en utilisant des filtres sur les attributs pour limiter la recherche. Le client peut alors se connecter en utilisant les adresses obtenues.

31.8.7.2. Via la technologie Jini

La technologie Jini offre une architecture logicielle pour créer et déployer des services au travers du réseau. Jini offre bien sûre un registre qui contient les services.

L'agent JMX doit enregistrer chacun de ces connecteurs auprès du registre de Jini et lui fournissant un objet de type stub et des attributs obligatoires et éventuellement optionnels pour qualifier le connecteur.

Le client JMX interroge le registre de SLP pour obtenir les stubs éventuellement en utilisant des filtres sur les attributs pour limiter la recherche. Le client peut alors se connecter en utilisant les stubs obtenus.

31.8.7.3. Via un annuaire et la technologie JNDI

JNDI est une API standard qui permet d'interagir avec différents services de nommage ou annuaires.

L'agent JMX doit enregistrer chacun de ces connecteurs auprès du registre de l'annuaire en lui fournissant son adresse et des attributs obligatoires et éventuellement optionnels pour qualifier le connecteur.

Le client JMX interroge le registre de l'annuaire pour obtenir les adresses éventuellement en utilisant des filtres sur les attributs pour limiter la recherche. Le client peut alors se connecter en utilisant les adresses obtenues.

31.9. Les notifications

Tous les types de MBeans peuvent émettre des notifications pour informer de certains événements survenus sur la ressource gérée par le MBean ou par le MBean lui-même.

Les notifications peuvent avoir plusieurs utilités : changement d'une valeur ou de l'état d'un attribut, signaler un événement ou un problème, ...

31.9.1. L'interface NotificationBroadcaster

Pour émettre des notifications, un MBean peut implémenter l'interface NotificationBroadcaster mais son utilisation n'est plus recommandée. Il est préférable d'utiliser son interface fille NotificationEmitter.

L'interface NotificationBroadcaster définie trois méthodes qui permettent l'abonnement d'un listener, d'obtenir des informations sur les notifications et le désabonnement d'un listener.

Méthode	Rôle
void addNotificationListener(NotificationListener, NotificationFilter, Object)	Abonner le listener fourni en paramètre
MBeaNotificationInfo[] getNotificationInfo()	Renvoie un tableau contenant des informations sur les notifications pouvant être émises
void removeNotificationListener(NotificationListener)	Désabonner le listener fourni en paramètre

31.9.2. L'interface NotificationEmitter

Pour émettre des notifications, un MBean doit de préférence implémenter l'interface NotificationEmitter. Celle ci hérite de l'interface NotificationBroadcaster.

L'interface NotificationEmitter ne définit qu'une seule méthode supplémentaire, qui est une surcharge de la méthode removeNotificationListener() :

Méthode	Rôle
void removeNotificationListener(NotificationListener, NotificationFilter, Object)	Désabonner le listener fourni en paramètre

31.9.3. La classe NotificationBroadcasterSupport

Le plus simple pour implémenter les notifications dans un MBean est de le faire hériter de la classe NotificationBroadcasterSupport en plus d'implémenter l'interface du MBean. La classe NotificationBroadcasterSupport implémente l' interface NotificationEmitter et propose la méthode sendNotification() qui permet l'émission de la notification qui lui est fournie en paramètre. Celle ci est envoyée à chaque listener abonné.

31.9.4. La classe javax.management.Notification

La classe Notification encapsule une notification émise par un MBean suite à un événement.

Une notification est donc une instance de la classe javax.management.Notification ou d'une de ces sous classes : AttributeChangeNotification, JMXConnectionNotification, MBeanServerNotification, MonitorNotification, RelationNotification ou TimerNotification

Chaque notification possède :

- une source (Source) : c'est le nom de l'objet (ObjectName) du MBean qui émet la notification ou l'instance du MBean
- un type (Type) : c'est une chaîne de caractères dont chaque mot est séparé par un caractère point
- un numéro de séquence (SequenceNumber) : sa valeur est arbitraire mais il est préférable de l'incrémenter à chaque émission

- un timestamp (TimeStamp): c'est la date/heure d'émission de la notification
- un message (Message) : une chaîne de caractères qui fournie une description de la notification
- des données (UserData) : collection de données de type HashTable

Les sous classes de la classe Notification peuvent avoir des attributs supplémentaires.

Lorsque le serveur de MBeans envoie la notification au client JMX, il transforme la source en son ObjectName si l'objet source est l'instance du MBean. En effet, le client JMX connait l'ObjectName mais ne possède pas la référence sur l'instance du MBean.

Chaque notification doit obligatoirement avoir un numéro de séquence.

Exemple :

```
    Notification notif = new AttributeChangeNotification(this,
        numeroSequence, System.currentTimeMillis(),
        "Modification de la valeur", "Valeur", "int", this.valeur, valeur);

    this.valeur = valeur;

    sendNotification(notif);
```

Pour émettre une notification, il faut instancier un objet de type Notification et appeler la méthode sendNotification() en lui passant en paramètre l'instance créée.

Il faut redéfinir la méthode getNotificationInfo() qui renvoie un tableau de type MBeanNotificationInfo contenant des données sur les notifications pouvant être émises.

Un objet de type MBeanNotificationInfo possède :

- un ou plusieurs types
- un nom
- une description

Le MBean devrait fournir une fonctionnalité pour lui permettre de fournir à un client les données incluses dans une notification sans passer par celle ci.

31.9.5. Un exemple de notifications

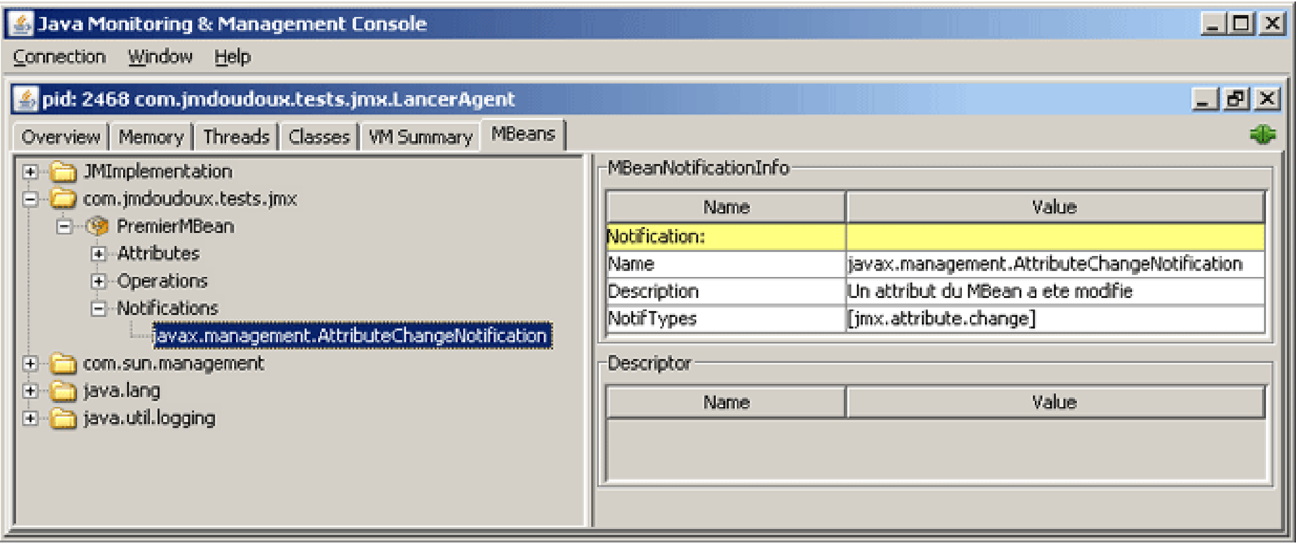
Cette section contient un exemple complet de mise en oeuvre de notifications par un MBean.

Exemple :

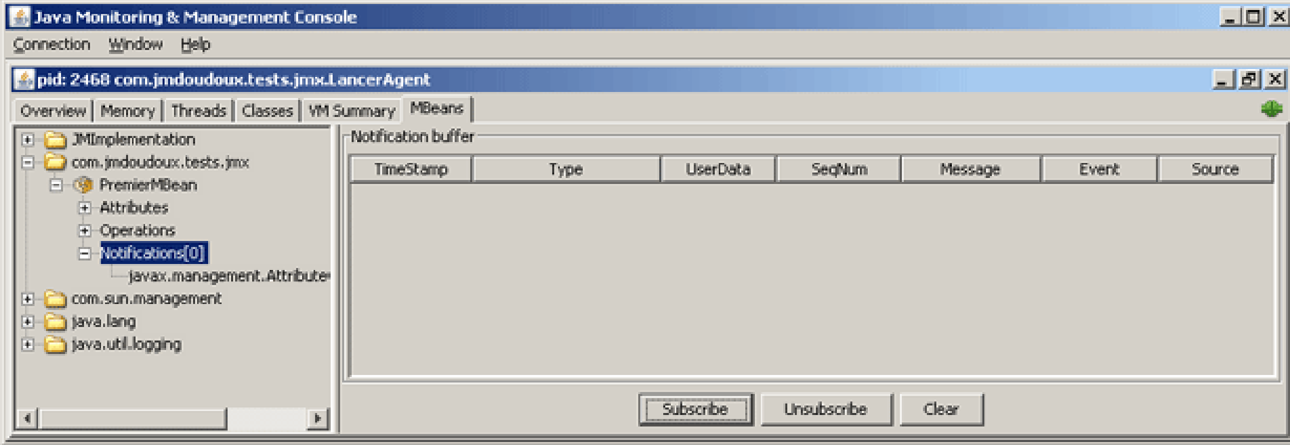
```
package com.jmdoudoux.tests.jmx;

import javax.management.AttributeChangeNotification;
import javax.management.MBeanNotificationInfo;
import javax.management.Notification;
import javax.management.NotificationBroadcasterSupport;

public class Premier extends NotificationBroadcasterSupport implements
    PremierMBean {

  private static String nom            = "PremierMBean";

  private int           valeur         = 100;

  private static long   numeroSequence = 0l;

  public String getNom() {
    return nom;
  }
```

```
  public int getValeur() {
    return valeur;
  }

  public synchronized void setValeur(int valeur) {

    numeroSequence++;
    Notification notif = new AttributeChangeNotification(this,
        numeroSequence, System.currentTimeMillis(),
        "Modification de la valeur", "Valeur", "int", this.valeur, valeur);

    this.valeur = valeur;

    sendNotification(notif);
  }

  public void rafraichir() {
    System.out.println("Rafraichir les donnees");
  }

  @Override
  public MBeanNotificationInfo[] getNotificationInfo() {
      String[] types = new String[] {
          AttributeChangeNotification.ATTRIBUTE_CHANGE
      };
      String name = AttributeChangeNotification.class.getName();
      String description = "Un attribut du MBean a ete modifie";
      MBeanNotificationInfo info =
          new MBeanNotificationInfo(types, name, description);
      return new MBeanNotificationInfo[] {info};
  }

}
```

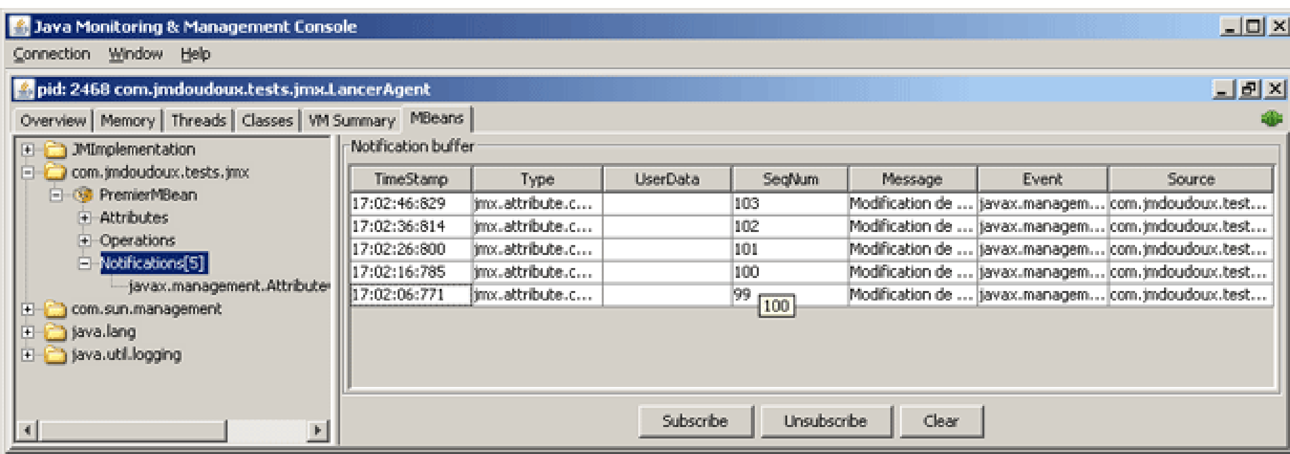
Il faut lancer l'agent et utiliser l'outil JConsole pour visualiser les notifications.



Il suffit de sélectionner Notifications et de cliquer sur le bouton «Subscribe»



Les notifications sont affichées au fur et à mesure de leur arrivée :



31.9.6. L'abonnement aux notifications par un client JMX

Un client JMX distant peut enregistrer un listener de type NotificationListener pour lui permettre d'être informer des changement du statut de la connexion utilisée pour les notifications. L'enregistrement se fait en utilisant la méthode addConnectionNotificationListener() sur une instance de l'interface JMXConnector.

Une classe d'un client JMX s'abonne aux notifications en enregistrant un listener de type NotificationListener en utilisant la méthode addNotificationListerner() sur l'instance de type MBeanServerConnection. Plusieurs listeners peuvent s'abonner à une même notification mais un même listener ne peut s'abonner qu'une seule fois à une notification.

L'interface NotificationListener ne définit qu'une seule méthode :
public void handleNotification(Notification notif, Object handback)

Cette méthode de type callback sera invoquée pour chaque listener abonné lors de l'émission d'une notification.

Il est possible de fournir un objet de type NotificationFilter dont l'implémentation encapsule un filtre sur les notifications que le client souhaite recevoir. L'interface NotificationFilter ne définit qu'une seule méthode isNotificationEnabled(Notification) qui renvoie un booléen.

JMX propose trois filtres en standard :

- AttributeChangeNotificationFilter : filtre sur les notifications de type AttributeChangeNotification
- MBeanServerNotificationListener : filtre sur les notifications de type MBeanServerNotification (hérite de la classe NotificationFilterSupport)
- NotificationFilterSupport : filtre sur l'attribut type des MBeans qui émettent les notifications

31.10. Les Dynamic MBeans

Un MBean standard expose ces fonctionnalités au travers d'une interface statique. Ainsi une propriété est définie à l'aide d'un getter et/ou d'un setter. Les MBeans standards ne permettent pas de répondre à tous les besoins notamment lorsque l'interface du MBean ne peut pas être définie de façon statique.

Parfois l'interface ne peut être définie que de façon dynamique à l'exécution : c'est par exemple le cas si les attributs sont issues d'une collection de type Map ou de la lecture d'une ressource externe comme un fichier.

Les Dynamic MBeans sont donc utiles si le nombre de fonctionnalités susceptibles d'être invoquées varie au cours des exécutions.

Le développement d'un MBean dynamic est complexe et nécessite que le MBean implémente l'interface DynamicMBean dont les méthodes retournent des informations statiques dans l'implémentation. Il est nécessaire d'utiliser et d'assembler des structures d'informations qui sont parfois redondantes. Ces informations sont encapsulées dans plusieurs classes du package javax.management : MBeanInfo, MBeanAttributeInfo, MBeanOperationInfo, ...

Les Dynamic MBeans ne possèdent pas un getter et un setter pour chaque attribut : ils proposent à la place des méthodes génériques pour obtenir et mettre à jour la valeur d'un attribut et invoquer les méthodes du MBean dynamiquement à partir de son nom.

Les fonctionnalités exposées par le MBean sont contenues dans un objet de type MBeanInfo retourné par la méthode getMBeanInfo() de l'interface DynamicMBean. Ces fonctionnalités concernent les attributs, les opérations et les notifications : les informations qu'il est cependant possible d'obtenir sur le MBean et ses méthodes sont plus riches que celles d'un MBean standard.

Ainsi l'interface du MBean est statique mais son implémentation expose dynamiquement les fonctionnalités du MBean.

L'exploitation des MBeans standards et dynamic ne fait aucun différence pour les clients JMX qui utilisent l'un et l'autre de la même façon. La différence se fait dans la façon dont ils exposent chacun leurs fonctionnalités :

- une interface statique pour les MBeans standards
- une description dynamique pour les MBeans dynamic

Un agent JMX n'a pas besoin de faire de l'introspection sur un Dynamic MBean pour découvrir ces fonctionnalités, il lui suffit de lire ses méta données obtenues via la méthode getMBeanInfo().

31.10.1. L'interface DynamicMBean

L'interface javax.management.DynamicMBean propose des méthodes pour permettre à un MBean Dynamic d'exposer dynamiquement ses fonctionnalités. Celles ci sont exposés au moyen de descripteurs qui sont fournis grâce à ses méthodes.

Méthode	Rôle
MBeanInfo getMBeanInfo()	Renvoie un objet de type MbeanInfo qui encapsule les fonctionnalités exposées par le MBean
Object getAttribute(String attribute)	Permet d'obtenir la valeur d'un attribut à partir de son nom
void setAttribute(Attribute attribute)	Permet de mettre à jour la valeur d'un attribut
AttributeList getAttributes(String[] attributes)	Permet d'obtenir la valeur d'un ensemble d'attributs à partir de leur nom
AttributeList setAttributes(AttributeList attributes)	Permet de mettre à jour la valeur d'un ensemble d'attributs
Object invoke(String actionName, Object params[], String signature[])	Permet d'invoquer une opération

La classe MBeanInfo encapsule les fonctionnalités exposées par le MBean : une collection des attributs avec leur type et leur nom, une collection des constructeurs, une collection des opérations invocables avec leurs paramètres, une collection des notifications et quelques informations.

31.10.2. Les méta données d'un Dynamic MBean

La méthode getMBeanInfo() renvoie un objet de type MBeanInfo qui encapsule les méta données des fonctionnalités du MBean.

31.10.2.1. La classe MBeanInfo

La classe javax.management.MBeanInfo est une classe qui encapsule les méta données des fonctionnalités du MBean. Chaque instance est immuable.

Cette classe propose plusieurs méthodes pour obtenir les méta données selon le type de leurs fonctionnalités :

Méthode	Rôle
MBeanAttributeInfo[] getAttributes()	Renvoie un tableau de type MBeanAttributeInfo qui contient les méta données des attributs
MBeanConstructorInfo[] getConstructors()	Renvoie un tableau de type MBeanConstructorInfo qui contient les méta données des constructeurs
String getDescription()	Renvoie une description du MBean
MBeanNotificationInfo[] getNotifications()	Renvoie un tableau de type MBeanNotificationInfo qui contient les méta données des notifications
MBeanOperationInfo[] getOperations()	Renvoie un tableau de type MBeanOperationInfo qui contient les méta données des opérations

Elle possède un seul constructeur :

MBeanInfo(String className, String description, MBeanAttributeInfo[] attributes, MBeanConstructorInfo[] constructors, MBeanOperationInfo[] operations, MBeanNotificationInfo[] notifications)

31.10.2.2. La classe MBeanFeatureInfo

La classe javax.management.MBeanFeatureInfo est la classe mère des classes qui encapsulent les méta données d'une fonctionnalité du MBean.

Cette classe possède deux propriétés :

Propriété	Rôle
Description	Description de la fonctionnalité
Name	Nom de la fonctionnalité

Elle ne propose que des getters sur ces propriétés.

31.10.2.3. La classe MBeanAttributeInfo

La classe javax.management.MBeanAttributeInfo encapsule les méta données d'un attribut du MBean. Elle hérite de la classe MBeanFeatureInfo.

Elle possède plusieurs propriétés :

Propriété	Rôle
String type	type de l'attribut
boolean isReadable	indique si l'attribut est lisible
boolean isWritable	indique si l'attribut est modifiable
boolean isIs	indique si le getter est de type isXXX pour les booléens

Chaque instance de cette classe est immuable : elle ne propose donc que des getters sur ces propriétés.

Elle possède deux constructeurs :

- MBeanAttributeInfo(String name, String description, java.lang.reflect.Method getter, java.lang.reflect.Method setter) : les informations sur l'attribut sont obtenues par instrospection sur le getter et le setter
- MBeanAttributeInfo(String name, String type, String description, boolean isReadable, boolean isWritable, boolean isIs)

31.10.2.4. La classe MBeanParameterInfo

La classe javax.management.MBeanParameterInfo encapsule les méta données d'un paramètre d'un constructeur ou d'une méthode du MBean. Elle hérite de la classe MBeanFeatureInfo.

Elle possède une propriété :

Propriété	Rôle
String type	le type du paramètre

Chaque instance de cette classe est immuable : elle ne propose donc que des getters sur ces propriétés.

Elle possède un seul constructeur :

MBeanParameterInfo(java.lang.String name, java.lang.String type, java.lang.String description)

31.10.2.5. La classe MBeanConstructorInfo

La classe javax.management.MBeanConstructorInfo encapsule les méta données d'un constructeur du MBean. Elle hérite de la classe MBeanFeatureInfo.

Elle possède une propriété :

Propriété	Rôle
MbeanParameterInfo[] signature	renvoie un tableau des paramètres du constructeur

Chaque instance de cette classe est immuable : elle ne propose donc que des getters sur ces propriétés.

Elle possède deux constructeurs :

- MBeanConstructorInfo(String description, java.lang.reflect.Constructor constructor) : les informations sur le constructeur sont obtenues par instrospection sur celui fourni en paramètre
- MBeanConstructorInfo(String name, String description, MBeanParameterInfo[] signature)

31.10.2.6. La classe MBeanOperationInfo

La classe javax.management.MBeanOperationInfo encapsule les méta données d'une méthode du MBean. Elle hérite de la classe MBeanFeatureInfo.

Elle possède plusieurs propriétés :

Propriété	Rôle
MbeanParameterInfo[] signature	renvoie un tableau des paramètres de la méthode
int impact	précise la natute de la méthode : les valeurs possibles sont INFO, ACTION, ACTION_INFO, UNKNOWN
String returnType	type de la valeur de retour de la méthode

Chaque instance de cette classe est immuable : elle ne propose donc que des getters sur ces propriétés.

Elle possède deux constructeurs :

- MBeanOperationInfo(String description, java.lang.reflect.Method method) : les informations sur la méthode sont obtenues par instrospection sur celles fournies en paramètre
- MBeanOperationInfo(String name, String description, MBeanParameterInfo[] signature, String type, int impact)

La classe MbeanOperationInfo définit plusieurs constantes utilisables pour sa propriété impact :

Constante	Rôle
INFO	précise que la méthode ne fait que lire l'état du MBean
ACTION	précise que la méthode va modifier l'état du MBean
ACTION_INFO	précise que la méthode lit et modifie l'état du MBean
UNKNOWN	précise que la nature de la méthode est inconnue

31.10.2.7. La classe MBeanNotificationInfo

La classe javax.management.MBeanNotificationInfo encapsule les méta données d'une notification du MBean. Elle hérite de la classe MBeanFeatureInfo.

Elle possède une propriété :

Propriété	Rôle
String[] notifTypes	renvoie un tableau du libellé des types de la notification (ce n'est pas le nom des classes)

Chaque instance de cette classe est immuable : elle ne propose donc que des getters sur ses propriétés.

Elle possède un seul constructeur :

MBeanNotificationInfo(java.lang.String[] notifTypes, java.lang.String name, java.lang.String description)

31.10.3. La définition d'un MBean Dynamic

Les Dynamic MBeans proposent des fonctionnalités plus avancées que les MBeans standards mais ils sont aussi plus complexes à mettre à oeuvre.

Un Dynamic MBean n'a pas besoin de respecter des conventions de nommage particulières ni de définir sa propre interface mais simplement d'implémenter l'interface DynamicMBean et avoir au moins un constructeur public.

Ce premier exemple est une implémentation en MBean Dynamic du MBean standard PremierMBean définit précédemment.

Exemple :

```
package com.jmdoudoux.tests.jmx;

import java.util.Iterator;

import javax.management.Attribute;
import javax.management.AttributeList;
import javax.management.AttributeNotFoundException;
import javax.management.DynamicMBean;
import javax.management.InvalidAttributeValueException;
import javax.management.MBeanAttributeInfo;
import javax.management.MBeanConstructorInfo;
import javax.management.MBeanException;
import javax.management.MBeanInfo;
import javax.management.MBeanOperationInfo;
import javax.management.MBeanParameterInfo;
import javax.management.ReflectionException;

public class PremierDynamic implements DynamicMBean {

  private static String nom = "PremierDynamic";
  private int valeur = 100;

  public PremierDynamic() {

  }

  @Override
  public Object getAttribute(String attribute)
      throws AttributeNotFoundException, MBeanException, ReflectionException {
    Object resultat = null;

    if (attribute.equals("nom")) {
      resultat = getNom();
    } else {

      if (attribute.equals("valeur")) {
        resultat = getValeur();
      } else {
        throw new AttributeNotFoundException(attribute);
      }
    }
    return resultat;

  }

  @Override
  public AttributeList getAttributes(String[] attributes) {
    AttributeList attributs = new AttributeList();
    attributs.add(new Attribute("nom", getNom()));
    attributs.add(new Attribute("valeur", getValeur()));
    return attributs;
```

```
  }

  @Override
  public MBeanInfo getMBeanInfo() {
    MBeanParameterInfo[] sansParamInfo = new MBeanParameterInfo[0];

    MBeanAttributeInfo attributs[] = new MBeanAttributeInfo[2];
    attributs[0] = new MBeanAttributeInfo("valeur", "int",
        "Valeur de l'instance", true, true, false);
    attributs[1] = new MBeanAttributeInfo("nom", "java.lang.String",
        "Nom de l'instance", true, false, false);

    MBeanConstructorInfo[] constructeurs = new MBeanConstructorInfo[1];
    constructeurs[0] = new MBeanConstructorInfo("PremierDynamic",
        "Constructeur par défaut de la classe", sansParamInfo);

    MBeanOperationInfo[] operations = new MBeanOperationInfo[1];
    operations[0] = new MBeanOperationInfo("rafraichir",
        "Rafraichir les données", sansParamInfo, void.class.getName(),
        MBeanOperationInfo.ACTION);

    return new MBeanInfo(getClass().getName(), "Mon premier MBean Dynamic",
        attributs, constructeurs, operations, null);

  }

  @Override
  public Object invoke(String actionName, Object[] params, String[] signature)
      throws MBeanException, ReflectionException {
    try {
      if (actionName.equals("rafraichir")) {
        rafraichir();
      }
      return null;
    } catch (Exception x) {
      throw new MBeanException(x);
    }

  }

  @Override
  public void setAttribute(Attribute attribute)
      throws AttributeNotFoundException, InvalidAttributeValueException,
      MBeanException, ReflectionException {

    String name = attribute.getName();
    try {
      if (name.equals("valeur")) {
        setValeur(((Integer) attribute.getValue()).intValue());
      } else {
        throw new AttributeNotFoundException(name);
      }

    } catch (ClassCastException cce) {
      throw new InvalidAttributeValueException(name);
    }
  }

  @Override
  @SuppressWarnings("unchecked")
  public AttributeList setAttributes(AttributeList attributes) {
    for (Iterator  i = attributes.iterator(); i.hasNext();) {
      Attribute attr = (Attribute) i.next();
      try {
        setAttribute(attr);
      } catch (AttributeNotFoundException e) {
        e.printStackTrace();
      } catch (InvalidAttributeValueException e) {
        e.printStackTrace();
      } catch (MBeanException e) {
        e.printStackTrace();
      } catch (ReflectionException e) {
        e.printStackTrace();
      }
```

```
    }
    return attributes;

  }

  private String getNom() {
    return nom;
  }

  private int getValeur() {
    return valeur;
  }

  private synchronized void setValeur(int valeur) {
    this.valeur = valeur;
  }

  private void rafraichir() {
    System.out.println("Rafraichir les donnees");

  }

}
```

L'intérêt de cet exemple est purement pédagogique car dans ce cas aucune fonctionnalité dynamique n'est utilisée mais il illustre bien la complexité d'écriture d'un MBean Dynamic par rapport à son équivalent sous la forme d'un MBean standard.

Les Dynamic MBeans peuvent tous fournir une description détaillée de leurs fonctionnalités ce qui peut les rendre plus facile à exploiter.

Le second exemple utilise une collection pour stocker ces attributs : cette collection pourrait par exemple être remplie en lisant un fichier de configuration. L'implémentation sous la forme d'un MBean standard est impossible car il n'est pas possible de connaître le contenu de la collection en dehors du contexte d'exécution.

Exemple :

```
package com.jmdoudoux.tests.jmx;

import java.util.Hashtable;
import java.util.Iterator;

import javax.management.Attribute;
import javax.management.AttributeList;
import javax.management.AttributeNotFoundException;
import javax.management.DynamicMBean;
import javax.management.InvalidAttributeValueException;
import javax.management.MBeanAttributeInfo;
import javax.management.MBeanConstructorInfo;
import javax.management.MBeanException;
import javax.management.MBeanInfo;
import javax.management.MBeanOperationInfo;
import javax.management.MBeanParameterInfo;
import javax.management.ReflectionException;

public class SecondDynamic implements DynamicMBean {

  private Hashtable<String, Object> attributs = new Hashtable<String, Object>();

  public SecondDynamic() {
    attributs.put("attrString1", "string");
    attributs.put("attrInt1", 0);
    attributs.put("attrString2", "string");
    attributs.put("valeur", 0);
  }

  @Override
  public synchronized Object getAttribute(String attribute)
      throws AttributeNotFoundException, MBeanException, ReflectionException {
    Object resultat = null;
```

```
    if (attributs.containsKey(attribute)) {
      resultat = attributs.get(attribute);
    } else {
      throw new AttributeNotFoundException(attribute);
    }
    return resultat;
  }

  @Override
  public AttributeList getAttributes(String[] attributes) {
    AttributeList resultat = new AttributeList();

    for (String cle : attributes) {
      if (attributs.containsKey(cle)) {
        resultat.add(new Attribute(cle, attributs.get(cle)));
      }
    }
    return resultat;
  }

  @Override
  public MBeanInfo getMBeanInfo() {
    MBeanParameterInfo[] sansParamInfo = new MBeanParameterInfo[0];

    int i = 0;
    MBeanAttributeInfo attribs[] = new MBeanAttributeInfo[attributs.size()];
    for (String cle : attributs.keySet()) {
      attribs[i] = new MBeanAttributeInfo(cle, attributs.get(cle).getClass()
          .getName(), "Description de l'attribut " + cle, true, true, false);
      i++;
    }

    MBeanConstructorInfo[] constructeurs = new MBeanConstructorInfo[1];
    constructeurs[0] = new MBeanConstructorInfo("SecondDynamic",
        "Constructeur par défaut de la classe", sansParamInfo);

    MBeanOperationInfo[] operations = new MBeanOperationInfo[1];
    operations[0] = new MBeanOperationInfo("rafraichir",
        "Rafraichir les données", sansParamInfo, void.class.getName(),
        MBeanOperationInfo.ACTION);

    return new MBeanInfo(getClass().getName(), "Mon second MBean Dynamic",
        attribs, constructeurs, operations, null);

  }

  @Override
  public Object invoke(String actionName, Object[] params, String[] signature)
      throws MBeanException, ReflectionException {

    try {
      if (actionName.equals("rafraichir")) {
        rafraichir();
      }
      return null;
    } catch (Exception x) {
      throw new MBeanException(x);
    }

  }

  @Override
  public synchronized void setAttribute(Attribute attribute)
      throws AttributeNotFoundException, InvalidAttributeValueException,
      MBeanException, ReflectionException {

    String name = attribute.getName();

    if (attributs.containsKey(name)) {
      attributs.remove(name);
      attributs.put(name, attribute.getValue());
    } else {
      throw new AttributeNotFoundException(name);
```

```
    }
  }

  @Override
  @SuppressWarnings("unchecked")
  public synchronized AttributeList setAttributes(AttributeList attributes) {
    for (Iterator i = attributes.iterator(); i.hasNext();) {
      Attribute attr = (Attribute) i.next();
      try {
        setAttribute(attr);
      } catch (AttributeNotFoundException e) {
        e.printStackTrace();
      } catch (InvalidAttributeValueException e) {
        e.printStackTrace();
      } catch (MBeanException e) {
        e.printStackTrace();
      } catch (ReflectionException e) {
        e.printStackTrace();
      }
    }
    return attributes;

  }

  private void rafraichir() {
    System.out.println("Rafraichir les donnees");
  }

}
```

Il est possible d'avoir un contrôle précis sur les fonctionnalités exposées en fonction d'un contexte. Par exemple, il est possible en fonction de la valeur d'une propriété d'exposer toutes les fonctionnalités ou seulement un sous ensemble.

Remarque : il est tout à fait possible pour un MBean standard d'implémenter aussi l'interface DynamicMBean.

31.10.4. La classe StandardMBean

Il peut être intéressant pour un MBean standard de profiter de certaines fonctionnalités des MBeans Dynamic bien que l'interface du MBean soit statique. Ces fonctionnalités concernent par exemple la possibilité de fournir une description aux attributs ou aux méthodes.

Dans ce cas, plutôt que d'implémenter l'interface DynamicMBean, il est plus simple d'hériter de la classe javax.management.StandardMBean. Ainsi, les fonctionnalités du MBean sont toujours exposées sous la forme de son interface statique et il est possible de bénéficier des fonctionnalités des MBeans Dynamic.

La classe StandardMBean possède deux constructeurs :

Constructeur	Rôle
StandardMBean(Class interface)	Créer un MBean Dynamic à partir de l'interface du MBean
StandardMBean(Object implementation, Class interface)	Créer un MBean Dynamic à partir de l'instance du MBean et de son interface

Elle propose de nombreuses méthodes pour permettre d'interagir avec les informations dynamiques demandées sur une fonctionnalité.

La classe StandardMBean a été ajoutée par la version 1.2 de JMX.

31.11. Les Model MBeans

Les Model MBeans sont des Dynamic MBeans génériques et configurables.

Chaque implémentation de JMX à l'obligation de fournir une implémentation d'une classe nommée javax.management.modelmbean.RequiredModelMBean qui implémente l'interface ModelMBean

La classe RequiredModelMbean agit comme un modèle générique pour créer dynamiquement des MBeans à partir d'objets qui ne respectent pas les spécifications des MBeans. Un Model MBean est donc obligatoirement un Dynamic MBean puisqu'il n'est pas possible de connaître à l'avance la classe qu'il va encapsuler.

Les informations de configuration des fonctionnalités exposées sont encapsulées dans une instance de l'interface ModelMBeanInfo. Un descripteur encapsulé dans l'interface Descriptor permet de fournir le mapping entre une fonctionnalité exposée et la méthode correspondante à invoquer de l'instance de l'objet encapsulé dans le Model MBean.

Pour exposer un objet sous la forme d'un Model MBean, il faut suivre plusieurs étapes :

- instancier l'objet
- créer une instance de la classe RequiredModelMBean
- fournir les informations sur les fonctionnalités exposées au Model MBean
- fournir au ModelMBean l'instance de l'objet
- enregistrer le Model MBean dans le serveur de MBeans

Les Model MBeans sont une des fonctionnalités avancées proposées par la spécification JMX. Leur mise en oeuvre est relativement compliquée ce qui limite leur mise en oeuvre. Ils ont cependant plusieurs intérêts :

- ils permettent à un objet ne respectant pas les spécifications des MBeans d'être exposés au travers de JMX
- ils permettent d'ajouter une redirection qui assure aux clients JMX que la classe RequiredModelMbean est toujours présente puisque fournie obligatoirement avec l'implémentation alors que la classe qu'elle expose ne l'est pas forcement
- ils peuvent fournir des informations supplémentaires aux classes MBean*Info sous la forme d'objets de type Descriptor

31.11.1. L'interface ModelMBean et la classe RequiredModelMBean

Chaque implémentation de JMX doit fournir une implémentation de l'interface ModelMBean sous la forme d'une classe nommée RequiredModelMBean

L'interface ModelMBean doit être implémentée par un Model MBean. Cette interface définie deux méthodes :

Méthode	Rôle
void setModelMBeanInfo(ModelMBeanInfo inModelMBeanInfo)	Fournir les informations de configuration du Model MBean
void setManagedResource(java.lang.Object mr, java.lang.String mr_type)	Fournir l'instance du l'objet sur lequel le Model Bean va invoquer les méthodes

L'interface ModelMBeanInfo encapsule les informations et les descriptions des fonctionnalités de l'objet qui seront exposées au travers du Model MBean.

Chaque implémentation de JMX doit fournir une implémentation de la classe RequiredModelMBean. Cette implémentation doit fournir les fonctionnalités de base pour exposer une instance d'une classe sous la forme d'un MBean sans que cette classe respecte les spécifications de JMX.

La classe RequiredModelMBean possède deux constructeurs :

Constructeur	Rôle

RequiredModelMBean()	Créer une instance avec un objet de type ModelMBeanInfo vide
RequiredModelMBean(ModelMBeanInfo mbi)	Créer une instance avec l'objet de type ModelMBeanInfo fourni en paramètre

Les méthodes setModelMBeanInfo() et setManagedResource() peuvent être utilisées pour fournir respectivement la description des fonctionnalités exposées par le MBean et l'instance de la classe qui sera encapsulée par le MBean.

Pour des besoins spécifiques, il est possible de créer une classe fille de la classe RequiredModelMBean.

31.11.2. La description des fonctionnalités exposées

La classe javax.management.modelmbean.ModelMBeanInfoSupport propose une implémentation de l'interfaceMBeanInfo qui facilite la création d'une instance du type de cette interface.

Cette classe propose trois constructeurs :

Constructeur	Rôle
ModelMBeanInfoSupport(ModelMBeanInfo mbi)	Créer une instance qui est une duplication de celle fournie en paramètre
ModelMBeanInfoSupport(java.lang.String className, java.lang.String description, ModelMBeanAttributeInfo[] attributes, ModelMBeanConstructorInfo[] constructors, ModelMBeanOperationInfo[] operations, ModelMBeanNotificationInfo[] notifications)	Créer une instance avec les informations fournies en paramètre et un descripteur par défaut. Le premier paramètre est le nom pleinement qualifié de la classe qui sera encapsulée par le MBean
ModelMBeanInfoSupport(java.lang.String className, java.lang.String description, ModelMBeanAttributeInfo[] attributes, ModelMBeanConstructorInfo[] constructors, ModelMBeanOperationInfo[] operations, ModelMBeanNotificationInfo[] notifications, Descriptor mbeandescriptor)	Créer une instance avec les informations et le descripteur fournis en paramètre. Le premier paramètre est le nom pleinement qualifié de la classe qui sera encapsulée par le MBean

Les informations sur les fonctionnalités exposées (attributs, constructeurs, opérations et notifications) sont décrites grâce à différents objets :

- ModelMBeanAttributeInfo : décrit les informations relatives à un attributs
- ModelMBeanConstructorInfo : décrit les informations relatives à un constructeur
- ModelMBeanOperationInfo : décrit les informations relatives à une opération
- ModelMBeanNotificationInfo : décrit les informations relatives à une notification

L'interface Descriptor permet de fournir des informations supplémentaires sur un attribut, un constructeur, une opération ou une notification. Une instance de type Descriptor peut être associée à chaque instance des classes ModelMBean*Info.

Une instance de Descriptor encapsule une collection de champs ayant chacun la forme nom=valeur.

La classe DescriptorSupport implémente l'interface Descriptor et permet de facilement créer une instance de type Descriptor.

Pour créer une instance de l'interface ModelMBeanInfo, il faut écrire beaucoup de code car il faut créer au moins un objet pour chaque éléments exposés par le MBean.

Remarque importante : il faut définir chaque attribut avec un objet de type ModelMBeanAttributeInfo mais il faut aussi impérativement définir chaque getter et chaque setter dans un objet de type ModelMBeanOperationInfo. Ceci est nécessaire car les conventions de nommage des JavaBeans n'ont pasd'obligation a être respectée dans la classe qui sera

encapsulée par le MBean.

Un objet de type RequiredModelMBean ne fait pas d'introspection sur la classe qu'il encapsule pour vérifier les informations fournies dans le ModelMBeanInfo : il fait aveuglement confiance aux informations qu'il contient.

Certaines implémentations peuvent fournir des utilitaires pour faciliter l'instanciation de l'interface ModelMBeanInfo par exemple à partir de fichiers XML, ce qui réduit la quantité de code à produire pour développer un Model MBean.

31.11.3. Un exemple de mise en oeuvre

L'exemple de cette section va exposer sous la forme d'un Model MBean une instance d'une simple classe nommée MaClasse.

Exemple :

```
package com.jmdoudoux.tests.jmx;

public class MaClasse {

  private static String nom = "MaClasse";
  private int valeur = 100;

  public String getNom() {
    return nom;
  }

  public int getValeur() {
    return valeur;
  }

  public synchronized void setValeur(int valeur) {
     this.valeur = valeur;
  }

  public void rafraichir() {
    System.out.println("Rafraichir les donnees");
  }

  public MaClasse() {
  }
}
```

La classe MaClasse est un simple POJO qui n'implémente aucune interface particulière : elle ne respecte aucune spécification de JMX. L'exemple suivant va encapsuler une instance de cette classe dans un Model MBean et fournir une description de ses fonctionnalités exposées par le Model MBean.

Dans l'agent JMX, une instance de la classe RequiredModelMBean est instanciée en passant en paramètre de son constructeur l'instance de l'interface ModelMBeanInfo.

Les informations sur les fonctionnalités exposées par le Model MBean et le mapping avec les méthodes correspondantes à invoquer sont encapsulées dans cette instance de l'interface ModelMBeanInfo.

Exemple :

```
package com.jmdoudoux.tests.jmx;

import java.lang.management.ManagementFactory;

import javax.management.Descriptor;
import javax.management.InstanceAlreadyExistsException;
import javax.management.InstanceNotFoundException;
import javax.management.MBeanException;
import javax.management.MBeanParameterInfo;
import javax.management.MBeanRegistrationException;
import javax.management.MBeanServer;
```

```
import javax.management.MalformedObjectNameException;
import javax.management.NotCompliantMBeanException;
import javax.management.ObjectName;
import javax.management.RuntimeOperationsException;
import javax.management.modelmbean.DescriptorSupport;
import javax.management.modelmbean.InvalidTargetObjectTypeException;
import javax.management.modelmbean.ModelMBeanAttributeInfo;
import javax.management.modelmbean.ModelMBeanConstructorInfo;
import javax.management.modelmbean.ModelMBeanInfo;
import javax.management.modelmbean.ModelMBeanInfoSupport;
import javax.management.modelmbean.ModelMBeanOperationInfo;
import javax.management.modelmbean.RequiredModelMBean;

public class LancerAgentModelMBean {

  public static void main(String[] args) {
    MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();

    ObjectName name = null;
    try {
      name = new ObjectName(
          "com.jmdoudoux.tests.jmx:type=OpenMXBean,name=MaClasse");

      MaClasse maClasse = new MaClasse();
      RequiredModelMBean modelMBean = new RequiredModelMBean(creerMBeanInfo());
      modelMBean.setManagedResource(maClasse, "objectReference");
      mbs.registerMBean(modelMBean, name);

      System.out.println("Lancement ...");
      while (true) {
        Thread.sleep(1000);
      }
    } catch (MalformedObjectNameException e) {
      e.printStackTrace();
    } catch (NullPointerException e) {
      e.printStackTrace();
    } catch (InstanceAlreadyExistsException e) {
      e.printStackTrace();
    } catch (MBeanRegistrationException e) {
      e.printStackTrace();
    } catch (NotCompliantMBeanException e) {
      e.printStackTrace();
    } catch (InterruptedException e) {
    } catch (RuntimeOperationsException e) {
      e.printStackTrace();
    } catch (InstanceNotFoundException e) {
      e.printStackTrace();
    } catch (MBeanException e) {
      e.printStackTrace();
    } catch (InvalidTargetObjectTypeException e) {
      e.printStackTrace();
    }
  }

  private static ModelMBeanInfo creerMBeanInfo() {
    Descriptor descriptorValeur = new DescriptorSupport(new String[] {
        "name=Valeur", "descriptorType=attribute", "default=0",
        "displayName=Valeur stockée dans la classe", "getMethod=getValeur",
        "setMethod=setValeur" });

    Descriptor descriptorNom = new DescriptorSupport(new String[] { "name=Nom",
        "descriptorType=attribute", "displayName=Nom de la classe",
        "getMethod=getNom" });

    ModelMBeanAttributeInfo[] mmbai = new ModelMBeanAttributeInfo[2];
    mmbai[0] = new ModelMBeanAttributeInfo("Valeur", "java.lang.Integer",
        "Valeur stockée dans la classe", true, true, false, descriptorValeur);
    mmbai[1] = new ModelMBeanAttributeInfo("Nom", "java.lang.String",
        "Nom de la classe", true, false, false, descriptorNom);

    ModelMBeanOperationInfo[] mmboi = new ModelMBeanOperationInfo[4];

    mmboi[0] = new ModelMBeanOperationInfo("getValeur",
        "getter pour l'attribut Valeur", null, "Integer",
```

```
        ModelMBeanOperationInfo.INFO);

    MBeanParameterInfo[] mbpiSetValeur = new MBeanParameterInfo[1];
    mbpiSetValeur[0] = new MBeanParameterInfo("valeur", "java.lang.Integer",
        "valeur de l'attribut");
    mmboi[1] = new ModelMBeanOperationInfo("setValeur",
        "setter pour l'attribut Valeur", mbpiSetValeur, "void",
        ModelMBeanOperationInfo.ACTION);

    mmboi[2] = new ModelMBeanOperationInfo("getNom",
        "getter pour l'attribut Nom", null, "String",
        ModelMBeanOperationInfo.INFO);

    mmboi[3] = new ModelMBeanOperationInfo("rafraichir",
        "Rafraichir les données", null, "void", ModelMBeanOperationInfo.ACTION);

    ModelMBeanConstructorInfo[] mmbci = new ModelMBeanConstructorInfo[1];
    mmbci[0] = new ModelMBeanConstructorInfo("MaClasse",
        "Constructeur par défaut", null);

    return new ModelMBeanInfoSupport("com.jmdoudoux.tests.jmx.MaClasse",
        "Exemple de ModelBean", mmbai, mmbci, mmboi, null);
  }
}
```

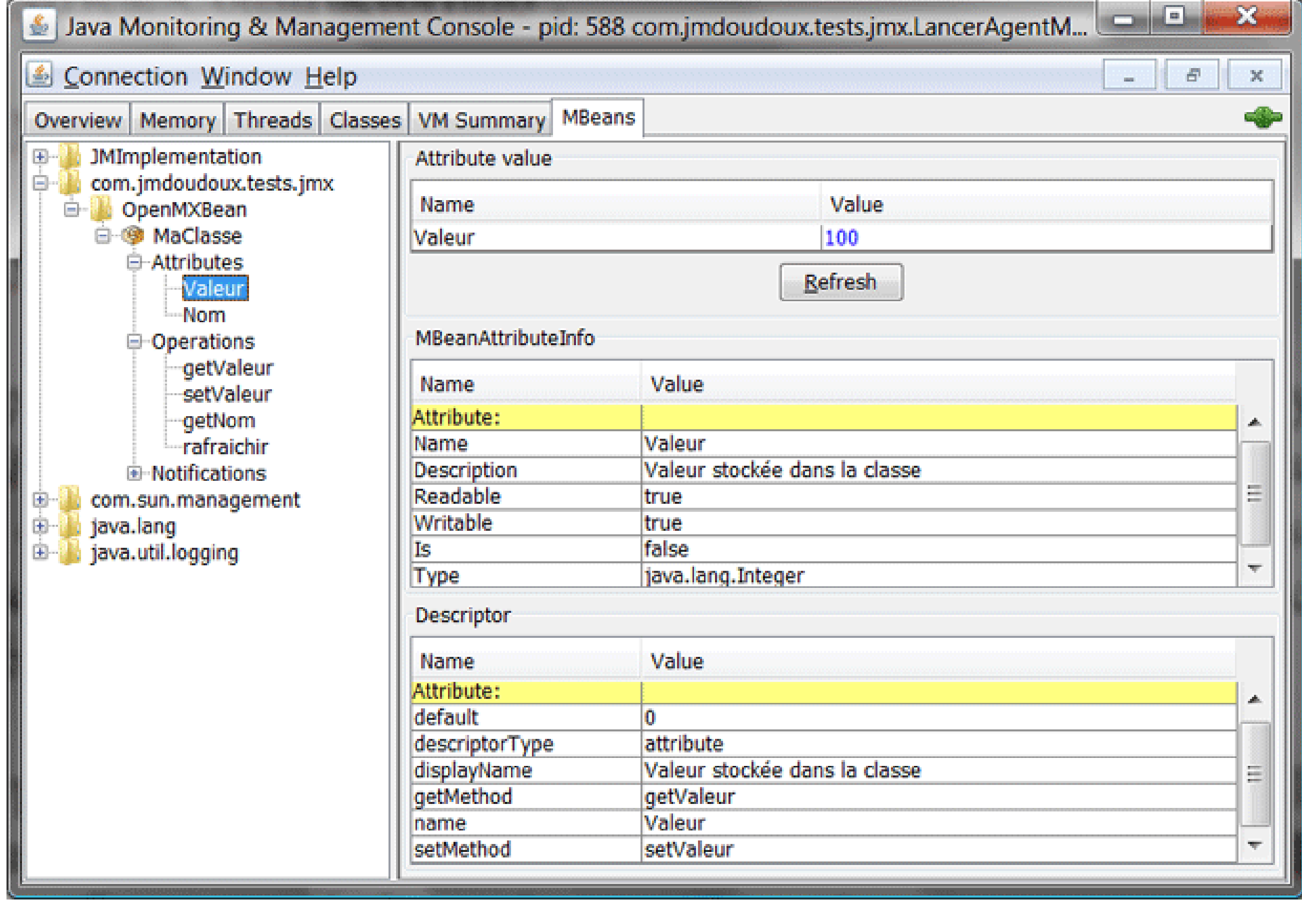
La partie la plus délicate lors de la mise en oeuvre d'un Model MBean est de créer cette instance de l'interface ModelMBeanInfo qui va contenir la description des fonctionnalités à exposer ainsi que le mapping vers les méthodes de l'instance à invoquer. Pour rendre le code plus lisible, la création de cette instance est faite dans une méthode dédiée.

L'instance de la classe à invoquer est fournie en paramètre du Model MBean en utilisant la méthode setManagedRessource(). Elle attend deux paramètres :

- l'instance de l'objet que le MBean va encapsuler
- une chaîne de caractères qui précise la typologie de référence fournie. Plusieurs valeurs sont utilisables : ObjectReference, Handle, IOR, EJBHandle ou RMIReference. Si la valeur fournie ne fait pas partie de cette liste, une exception de type InvalidTargetTypeException est levée

C'est bien le Model MBean est qui enregistré dans le serveur de MBeans. Pour les clients JMX, le Model MBean est un Dynamic MBean. Le client n'a pas besoin d'avoir la classe de l'instance encapsulée dans le Model MBean.

L'objet de type RequiredModelMBean va dynamiquement générer les entités nécessaires pour exposer les fonctionnalités sous la forme d'un Dynamic MBean.



Il suffit de lancer l'agent et d'ouvrir un client JMX pour pouvoir interagir avec le ModelMBean.

Hormis le fait que les getter et les setter apparaissent dans les opérations, rien ne distingue le ModelMBean d'un autre MBean pour le client JMX.

31.11.4. Les fonctionnalités optionnelles des Model MBeans

La classe RequiredModelMBean peut proposer un support optionnel de certaines fonctionnalités comme le logging, la persistance ou un cache de données (caching).

La fonctionnalité de cache de données permet de stocker dans une variable du MBean la valeur d'un attribut et de la renvoyer durant sa durée de vie dans le cache plutôt que de toujours solliciter dynamiquement l'instance encapsulée.

Certaines de ces fonctionnalités peuvent être configurées au travers des données fournies via un Descriptor. Celles ci peuvent être modifiées dynamiquement par le MBean ou par un client JMX pour adapter le comportement des fonctionnalités par défaut.

Pour connaître les fonctionnalités optionnelles implémentées et la façon de les mettre en oeuvre il faut consulter la documentation de l'implémentation utilisée. Leurs utilisations limitent donc la portabilité vers une autre implémentation de JMX.

31.11.5. Les différences entre un Dynamic MBean est un Model MBean

Un Model MBean est un Dynamic MBean mais il est plus souple à mettre en oeuvre : le code d'un Dynamic MBean doit être intégralement écrit alors que pour un Model MBean une implémentation par défaut est fournie obligatoirement par l'implémentation de JMX et il suffit de lui fournir les fonctionnalités exposées et une référence sur l'objet à invoquer.

Les traitements d'un Dynamic MBean doivent être codés dans le MBean alors qu'avec un Model MBean le code est contenu dans l'instance de la classe dont les fonctionnalités sont exposées par le MBean.

Les fonctionnalités exposées par un Dynamic MBean doivent être connues au moment de l'écriture du code du MBean. Avec un Model MBean, les fonctionnalités exposées peuvent être définies et modifiées dynamiquement en utilisant la méthode setModelMBeanInfo().

Un Model MBean peut fournir des descriptions complémentaires des fonctionnalités qu'il expose sous la forme d'objets de type Descriptor qui encapsulent des champs. Ces champs peuvent être personnalisés.

L'implémentation d'un Model MBean peut proposer des services optionnels (persistance, logging, caching) qui peuvent être dynamiquement configurés avec des champs d'un Descriptor.

31.12. Les Open MBeans

La spécification JMX permet l'utilisation de types complexes qui soient portables en utilisant les Open MBeans.

Un Open MBean est un Dynamic MBean ou tous les types utilisés doivent appartenir à une liste précisément définie dans les spécifications JMX. Ces types de base peuvent être utilisés dans un type composé dédié appelé Open Type.

Seul un sous ensemble restreint de classes peuvent être décrit sous la forme d'un OpenType :

- les wrapper de primitives : Integer, Boolean, Void, ...
- les classes String, BigDecimal, BigInteger, Date
- CompositeData et TabularData

Il faut noter que les types primitifs ne sont pas autorisés : il faut utiliser leur wrapper.

Ceci permet à un Open MBean d'être portable avec des types complexes mais sans utiliser un type spécifique qui devrait être fourni à chaque client JMX. Les Open MBeans sont ainsi les MBeans les plus ouverts et les plus portables avec les clients JMX puisqu'ils n'ont pas besoin d'ajouter dans leur classpath des classes spécifiques.

La restriction des types de données utilisables est aussi intéressante pour les connecteurs et les adaptateurs de protocole qui n'ont qu'à savoir gérer ces types.

Un objet de type OpenMBeanInfo encapsule la description des fonctionnalités exposées par un Open MBean en incluant en plus du type Java un objet de type OpenType pour chaque attributs et opérations.

Les spécifications des Open MBeans dans la version 1.0 de JMX est incomplète donc inutilisable. Dans la version 1.1, les spécifications sont complètes mais leur implémentation est facultative. A partir de la version 1.2, ils sont obligatoires dans toutes les implémentations.

31.12.1. La mise en oeuvre d'un Open MBean

Un Open MBean doit implémenter l'interface DynamicMBean et n'a pas besoin d'implémenter une interface spécifique aux Open MBeans. La différence avec un Dynamic MBean se fait à deux niveaux :

- une restriction forte sur les types de données utilisables par le MBean
- une description des fonctionnalités du MBean avec des interfaces dédiées aux Open MBeans

La description des fonctionnalités d'un Open MBean est assurée grâce aux interfaces du package javax.management.openmbean.

Pour distinguer les Open MBeans des autres MBeans, les données de description des fonctionnalités exposées sont encapsulées dans des interfaces dédiées du package javax.management.openmbean :

- OpenMBeanInfo : contient l'ensemble des fonctionnalités exposées

- OpenMBeanOperationInfo : contient la description d'une méthode
- OpenMBeanConstructorInfo : contient la description d'un constructeur
- OpenMBeanParameterInfo : contient la description d'un paramètre
- OpenMBeanAttributeInfo : contient la description d'un attribut

Chacune de ces interfaces possède une classe correspondante dont le nom se termine par Support.

Un objet de type OpenMBeanInfo peut décrire la valeur par défaut et les valeurs autorisées pour un attribut, un paramètre et une valeur de retour.

Pour la description des notifications, les Open MBeans utilisent la classe MBeanNotificationInfo.

L'interface OpenMBeanOperationInfo possède une propriété impact de type int qui précise l'impact de la méthode lorsqu'elle est invoquée. Les valeurs possibles sont :

- MBeanOperationInfo.INFO : la méthode est en lecture seule
- MBeanOperationInfo.ACTION : la méthode va modifier l'état du MBean
- MBeanOperationInfo.ACTION_INFO : la méthode en lecture/modification
- MBeanOperationInfo.UNKNOWN : la typologie d'action n'est pas précisée

31.12.2. Les types de données utilisables dans les Open MBeans

Les types utilisés pour les attributs, les paramètres et les valeurs de retour des opérations d'un Open MBean doivent appartenir à la liste des types précisés dans les spécifications JMX :

- les wrappers sur les types primitifs : java.lang.Boolean, java.lang.Byte, java.lang.Character, java.lang.Short, java.lang.Integer, java.lang.Long, java.lang.Float, java.lang.Double, java.lang.String, java.lang.Void, java.math.BigInteger, java.math.BigDecimal
- les classes de type de données standards : String, Date
- des classes de JMX : CompositeData, TabularData, et ObjectName

L'ensemble de ces types est désigné sous le nom Open Types.

31.12.2.1. Les Open Types

Les Open Types encapsulent la description d'un type de données utilisé par les Open MBeans. Ils permettent de fournir un moyen standard de décrire les types utilisables par les Open MBeans. Un Open Type peut représenter un type primitif sous la forme de leur wrapper, un type complexe composé d'autres Open Types ou un tableau d'Open Types.

La classe Abstraite OpenType est la classe mère des différentes classes qui encapsulent des Open Types :

- SimpleType : permet de décrire un type simple
- CompositeType : permet de décrire un type composé d'autres Open Types
- TabularType : permet de décrire un type de données tabulaires
- ArrayType : permet de décrire un type de données sous la forme d'un tableau multi-dimension

Les classes Composite Type, TabularType et ArrayType peuvent contenir d'autres Open Types.

Les instances des Open Types sont selon leur type une instance de différents types :

- un wrapper pour les types primitifs
- une instance de la classe String, Date, ou ObjectName
- une instance de la classe CompositeData
- une instance de la classe TabularData

Les interfaces CompositeData et TabularData encapsulent les données de leur type complexe respectif.

31.12.2.2. La classe CompositeType et l'interface CompositeData

Les données utilisés par un Open MBean peuvent être de type CompositeData. Cette classe permet d'encapsuler un objet complexe qui n'utilisera que des OpenTypes et respectera ainsi les spécifications des Open MBeans.

L'OpenType correspondant dans l'OpenMBeanInfo est CompositeType. Un CompositeType décrit un ensemble d'éléments ou de champs. Chaque élément possède un nom et un Open Type.

La classe CompositeData associe une clé à une valeur pour chacun des attributs définis avec un CompositeType. Un ou plusieurs éléments d'un CompositeData forme la clé de l'élément.

Les classes CompositeDataSupport et TabularDataSupport proposent une implémentation respective des interfaces CompositeData et TabularData pour faciliter la création d'une de leur instance.

Un objet de type CompositeData est immuable : toutes les données qu'il encapsule doivent donc être fournies lors de son instanciation. La classe CompositeDataSupport propose pour cela deux constructeurs :

Constructeur	Rôle
CompositeDataSupport(CompositeType compositeType, Map<String,?> items)	Créer une instance qui encapsule les données du CompositeType fourni avec les valeurs fournies dans la collection de type Map
CompositeDataSupport(CompositeType compositeType, String[] itemNames, Object[] itemValues)	Créer une instance qui encapsule les données du CompositeType avec les valeurs fournies sous la forme de deux tableaux (un qui encapsule le nom des attributs et l'autre qui encapsule leurs valeurs : l'ordre des données de deux tableaux doit correspondre).

La méthode getCompositeType() renvoie une instance de la classe CompositeType qui encapsule la description de l'Open Type encapsulé.

31.12.2.3. La classe TabularType et l'interface TabularData

Les données utilisés par un Open MBean peuvent de type TabularData. Ce classe permet d'encapsuler un tableau d'objets de type CompositeData qui n'utilisera que des OpenTypes et respectera ainsi les spécifications des Open MBeans.

L'OpenType correspondant dans l'OpenMBeanInfo est TabularType. Tous les éléments d'un TabularData possède le même CompositeType.

Une instance de TabularData encapsule une collection d'objets de type CompositeData. Chaque objet de ce type encapsule les données d'une occurrence. Chaque occurrence possède une clé unique obtenue à partir des données encapsulées dans le CompositeData.

Avec une instance de TabularData, il est possible d'ajouter ou de supprimer une ou plusieurs occurrences.

La classe TabularDataSupport encapsule un tableau à une dimension d'objets de type CompositeData. Tous les CompositeData contenu dans le TabularData doivent avoir le même CompositeType.

Chaque occurrence est associée à une clé unique qui identifie chaque occurrence. Cette clé est composée d'un ou plusieurs attributs encapsulés dans le CompositeData correspondant. Généralement cette clé est composée d'une seule donnée de type Integer ou String.

La classe TabularDataSupport propose toutes les méthodes nécessaires pour manipuler les occurrences qu'il encapsule : put(), putAll(), get(), remove(), clear(), size(), isEmpty(), containsKey(), keySet(), values(), ...

La méthode get() attend en paramètre un tableau des valeurs qui compose la clé et renvoie l'objet de type CompositeData associé si celui-ci existe.

31.12.3. Un exemple d'utilisation d'un Open MBean

La suite de cette section sera développée dans une version future de ce document

31.12.4. Les avantages et les inconvénients des Open MBeans

Le grand avantage des Open MBeans est de garantir l'interopérabilité.

Leur principal défaut est d'être un Dynamic MBean, donc relativement difficile à coder. Une alternative est de coder un MBean Standard qui n'utilise que des Open Types.

La version 1.1 des spécifications de JMX ne fournit pas tous les détails concernant les Open MBeans, leur support est donc optionnel. D'ailleurs l'implémentation de référence de la version 1.1 ne propose pas d'implémentation pour les Open MBeans.

31.13. Les MXBeans

La version 6.0 de Java introduit un nouveau type de MBean : les MXBeans. Ce type de MBeans permet d'utiliser des types définis par l'utilisateur du moment que ces types respectent les contraintes définis dans les spécifications.

Ces types sont convertis vers des Open Types tel que SimpleType, CompositeType, ArrayType, TabularType, ... définis pour les Open MBeans. Les Open Types sont définis dans le package javax.management.openmbean.

Cela permet une utilisation du MBean sans que le client JMX n'ai besoin du jar qui contiennent la définition du MBean même si ce client est distant.

Les MXBeans sont définis grâce à une interface statique mais contrairement aux MBeans standards le nom de la classe qui implémente l'interface du MXBean est libre.

La plate-forme Java fournit plusieurs MXBeans regroupés dans le package java.lang.management. Le développeur peut aussi écrire ces propres MXBeans.

31.13.1. La définition d'un MXBean

L'interface d'un MXBean doit soit avoir un nom qui termine par convention par MXBean soit être annotée avec l'annotation @javax.management.MXBean. Dans ce dernier cas, le nom de l'interface est libre.

Exemple :

```
package com.jmdoudoux.tests.jmx;

public interface InfoMXBean {

  public InfoParametre getInfo();
```

```
  public void rafraichir();

}
```

L'implémentation du MXBean doit implémenter son interface.

Exemple :

```
package com.jmdoudoux.tests.jmx;

import java.util.Date;

public class Info implements InfoMXBean {

  @Override
  public InfoParametre getInfo() {
    return new InfoParametre("nom1", new Date(), 100l, "description1");
  }

  @Override
  public void rafraichir() {
    System.out.println("Appel de la méthode rafraichir()");
  }

}
```

31.13.2. L'écriture d'un type personnalisé utilisé par le MXBean

L'exemple utilise un type personnalisé qui est un simple bean. Le constructeur est annoté avec l'annotation @ConstructorProperties. Cette annotation permet d'associer chaque paramètre d'un constructeur à une propriété.

Exemple :

```
package com.jmdoudoux.tests.jmx;

import java.beans.ConstructorProperties;
import java.util.Date;

public class InfoParametre {

  private String nom;
  private String description;
  private Date dateCreation;
  private long taille;

  @ConstructorProperties( { "nom", "dateCreation", "taille", "description" })
  public InfoParametre(String pnom, Date pdateCreation, long ptaille,
      String pdescription) {
    super();
    this.nom = pnom;
    this.description = pdescription;
    this.dateCreation = pdateCreation;
    this.taille = ptaille;
  }

  public String getNom() {
    return nom;
  }

  public String getDescription() {
    return description;
  }

  public void setDescription(String description) {
    this.description = description;
  }
```

```
  public Date getDateCreation() {
    return dateCreation;
  }

  public void setDateCreation(Date dateCreation) {
    this.dateCreation = dateCreation;
  }

  public long getTaille() {
    return taille;
  }

  public void setTaille(long taille) {
    this.taille = taille;
  }

}
```

JMX va utiliser les getter pour créer une instance de CompositeData qui encapsule les données. Pour recréer une instance de la classe InfoParametre à partir d'une instance de CompositeDate, JMX utilise les informations fournies par l'annotation @ConstructorProperties.

31.13.3. La mise en oeuvre d'un MXBean

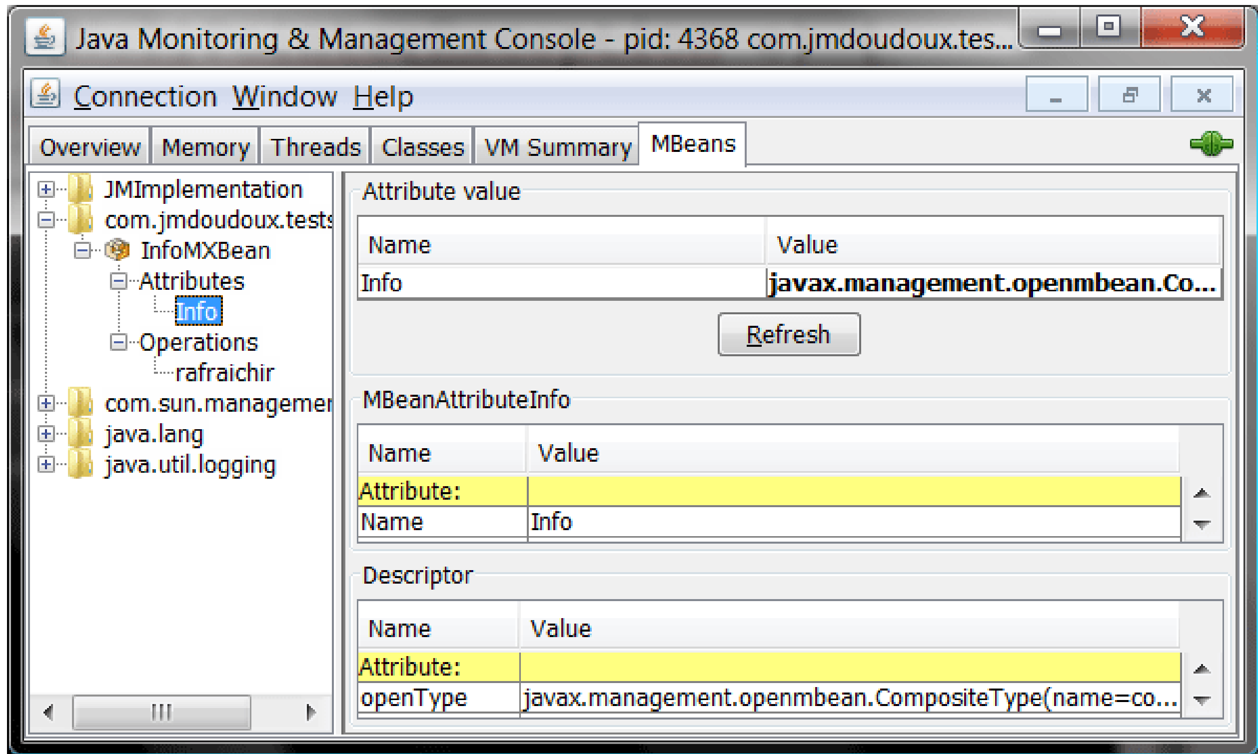
Le MXBean doit être instancié et enregistré dans le serveur de MBeans de la même manière que pour les autres MBeans.

Exemple :

```
package com.jmdoudoux.tests.jmx;

import java.lang.management.ManagementFactory;

import javax.management.InstanceAlreadyExistsException;
import javax.management.MBeanRegistrationException;
import javax.management.MBeanServer;
import javax.management.MalformedObjectNameException;
import javax.management.NotCompliantMBeanException;
import javax.management.ObjectName;

public class LancerAgentMXBean {

  public static void main(String[] args) {
    MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();

    ObjectName name = null;
    try {
      name = new ObjectName("com.jmdoudoux.tests.jmx:type=InfoMXBean");

      InfoMXBean mbean = new Info();

      mbs.registerMBean(mbean, name);

      System.out.println("Lancement ...");
      while (true) {

        Thread.sleep(1000);
      }
    } catch (MalformedObjectNameException e) {
      e.printStackTrace();
    } catch (NullPointerException e) {
      e.printStackTrace();
    } catch (InstanceAlreadyExistsException e) {
      e.printStackTrace();
    } catch (MBeanRegistrationException e) {
      e.printStackTrace();
    } catch (NotCompliantMBeanException e) {
```

```
            e.printStackTrace();
        } catch (InterruptedException e) {
        }
    }
}
```

La principale différence est au niveau du client JMX : l'attribut Info n'est du type InfoParametre mais de l'Open Type CompositeData.



En double cliquant sur la valeur de la ligne de l'attribut Info, il est possible d'afficher le détail des données encapsulées dans le CompositeData.

31.14. L'interface PersistentMBean

Il peut être nécessaire à un MBean de rendre persistant ses données. Dans ce cas, le MBean doit implémenter l'interface javax.management.PersistentMBean.

Cette interface ne définie que deux méthodes :

Méthode	Rôle
void load()	Lire des données du MBean à partir de l'unité de persistance
void store()	Ecriture des données du MBean à partir de l'unité de persistance

L'invocation de la méthode load() dans son constructeur est à la charge de l'implémentation du MBean.

L'invocation de la méthode save() peut être définie dans une Persistance Policy.

La suite de cette section sera développée dans une version future de ce document

L'implémentation du MBean est libre de choisir l'unité de persistance utilisée (fichier, base de données, ...)

31.15. Le monitoring d'une JVM

JMX est aussi utilisé pour surveiller et gérer la JVM en standard à partie de la version 5 de Java : un agent JMX peut être utilisé pour accéder à l'instrumentation de la JVM pour la surveiller et la gérer à distance.

Java 5.0 propose plusieurs fonctionnalités relatives au monitoring notamment :

- instrumentation de la JVM : la JVM est instrumentée avec des MBeans
- l'API de monitoring et de gestion : le package java.lang.management contient des classes et interfaces permettant d'obtenir des informations sur l'état de l'exécution de la JVM notamment sous la forme de MXBeans (mémoire, threads, garbage collection, ...)
- des outils : Java 5.0 propose l'outil JConsole qui permet d'afficher des informations sur la JVM et agit comme un client JMX graphique. Java 6.0 propose un outil encore plus évolué : Java Visual VM.

La JVM incorpore un serveur de MBeans et utilise des MXBeans dédiés fournis avec la plate-forme Java SE pour permettre de surveiller et gérer la JVM.

Ces MXBeans encapsulent chacun une grande fonctionnalité de la JVM : chargement des classes, ramasse miettes, compilateur JIT, threads, mémoire, ... Ceci permet d'obtenir de façon standard via JMX des informations sur la consommation en ressources et l'activité de la JVM.

Il est ainsi possible de consulter et d'interagir avec ces fonctionnalités en utilisant un client JMX comme JConsole fourni avec le JDK.

Pour pouvoir activer la connexion RMI de l'agent JMX de la JVM, il faut utiliser la propriété com.sun.management.jmxremote de la JVM

Exemple :

Exemple :

```
java -Dcom.sun.management.jmxremote MonApplication
```

Pour permettre un accès à un client distant sans authentification, il faut fournir trois autres propriétés à la JVM.

Exemple :

```
com.sun.management.jmxremote.port=9999
com.sun.management.jmxremote.authenticate=false
com.sun.management.jmxremote.ssl=false
```

Le port précisé ne doit pas être déjà utilisé.

La JSR 174 (Monitoring and Management Specification for the Java Virtual Machine JVM) définit plusieurs MXBeans dans le package java.lang.management pour la gestion et le monitoring de la JVM.

31.15.1. L'interface ClassLoadingMXBean

Cet MXBean permet de surveiller et de gérer le système de chargement des classes de la JVM.

Une instance de l'interface ClassLoadingMXBean est obtenue en invoquant la méthode getClassLoadingMXBean() de la fabrique ManagementFactory.

L'ObjectName pour l'unique instance de cet MXBean est : java.lang:type=ClassLoading

Cet MXBean permet d'obtenir plusieurs informations :

- le nombre de classes chargées dans la JVM
- le nombre total de classes chargées depuis le lancement de la JVM
- le nombre total de classes déchargées depuis le lancement de la JVM

Il permet aussi d'activer ou non l'affichage dans la sortie standard d'informations sur les activités de classloading de la JVM.

Exemple :

```
package com.jmdoudoux.tests.jmx.mxbeans;

import java.lang.management.ClassLoadingMXBean;
import java.lang.management.ManagementFactory;

public class TestClassLoadingMXB {

  public static void main(String[] args) {

    ClassLoadingMXBean clBean = ManagementFactory.getClassLoadingMXBean();
    System.out.printf("Loaded class count : %d\n", clBean
        .getLoadedClassCount());
    System.out.printf("Total loaded class count : %d\n", clBean.getTotalLoadedClassCount());
    System.out.printf("Unloaded class count : %d\n", clBean
        .getUnloadedClassCount());
    System.out.printf("isVerbose : %b \n", clBean.isVerbose());
    System.out.println();
    clBean.setVerbose(true);
  }

}
```

Résultat :

```
Loaded class count : 334
Total loaded class count : 422
Unloaded class count : 0
isVerbose : false

[Loaded java.util.IdentityHashMap$KeySet from shared objects file]
[Loaded java.util.IdentityHashMap$IdentityHashMapIterator from shared objects file]
[Loaded java.util.IdentityHashMap$KeyIterator from shared objects file]
[Loaded java.io.DeleteOnExitHook from shared objects file]
[Loaded java.util.HashMap$KeySet from shared objects file]
[Loaded java.util.LinkedHashMap$KeyIterator from shared objects file]
```

31.15.2. L'interface CompilationMXBean

Cet MXBean permet de surveiller le système de compilation JIT de la JVM.

Une instance de l'interface CompilationMXBean est obtenue en invoquant la méthode getCompilationMXBean() de la fabrique ManagementFactory.

L'ObjectName pour l'unique instance de cet MXBean est : java.lang:type=Compilation

Cet MXBean permet d'obtenir plusieurs informations :

- le nom du compilateur JIT utilisé par la JVM
- le temps estimé consommé pour la compilation de classes par le compilateur JIT
- un indicateur qui précise si la JVM permet le monitoring du compilateur JIT

Exemple :

```
package com.jmdoudoux.tests.jmx.mxbeans;

import java.lang.management.CompilationMXBean;
import java.lang.management.ManagementFactory;

public class TestCompilationMXB {

  public static void main(String[] args) {

    CompilationMXBean cBean = ManagementFactory.getCompilationMXBean();
    System.out.printf("Name : %s\n", cBean.getName());
    System.out.printf("Total compilation time : %d ms\n", cBean
        .getTotalCompilationTime());
    System.out.printf("iscompilationTimeMonitoringSupported : %b \n", cBean
        .isCompilationTimeMonitoringSupported());
  }
}
```

Résultat :

```
Name : HotSpot Client Compiler
Total compilation time : 7 ms
iscompilationTimeMonitoringSupported : true
```

31.15.3. L'interface GarbageCollectorMXBean

Cet MXBean permet de surveiller le ramasse miette de la JVM. Elle hérite de l'interface MemoryManagerMXBean. Une JVM peut avoir une ou plusieurs instances de cet MXBean, une pour chaque algorithme utilisé pour gérer la mémoire.

Les instances de l'interface GarbageCollectorMXBean sont obtenues en invoquant la méthode getGarbageCollectorMXBeans() de la fabrique ManagementFactory.

L'ObjectName d'une instance de cet MXBean est de la forme : java.lang:type=GarbageCollector, name=xxx

Cet MXBean permet d'obtenir plusieurs informations :

- le nombre de collecte qui ont été réalisées par l'algorithme
- le temps utilisé pour les collectes réalisées par l'algorithme

Exemple :

```
package
com.jmdoudoux.tests.jmx.mxbeans;
import
java.lang.management.GarbageCollectorMXBean;
import java.lang.management.ManagementFactory;
public class
TestGarbageCollectorMXB {
  public static void main(String[] args) {
    for (int i = 0; i < 100000; i++) {
      Byte[] tableau = new Byte[50 * 1024];
      if ((i % 10000) == 0) {
        System.out.print(".");
      }
    }
    System.out.println("");
    for (GarbageCollectorMXBean gcBean :
ManagementFactory
        .getGarbageCollectorMXBeans()) {
      System.out.printf("Memory manager
name : %s\n", gcBean.getName());
      System.out.printf("  isValid : %b\n", gcBean.isValid());
      for (String pool :
```

```
gcBean.getMemoryPoolNames()) {
        System.out.printf("  Memory pool name : %s\n", pool);
      }
      System.out.printf("\n  collectionCount : %d\n", gcBean
          .getCollectionCount());
      System.out.printf("  collectionTime : %d ms\n", gcBean
          .getCollectionTime());
      System.out.println();
    }
  }
}
```

Résultat :

```
..........
Memory manager name : Copy
  isValid : true
  Memory pool name : Eden
Space
  Memory pool name : Survivor Space
  collectionCount : 25000
  collectionTime : 1228 ms
Memory manager name : MarkSweepCompact
  isValid : true
  Memory pool name : Eden
Space
  Memory pool name :
Survivor Space
  Memory pool name : Tenured
Gen
  Memory pool name : Perm
Gen
  Memory pool name : Perm
Gen [shared-ro]
  Memory pool name : Perm
Gen [shared-rw]
  collectionCount : 0
  collectionTime : 0 ms
```

31.15.4. L'interface MemoryManagerMXBean

Cet MXBean permet de lister les gestionnaires de mémoire de la JVM. Une JVM peut avoir une ou plusieurs instances de cet MXBean, une pour chaque gestionnaire utilisé pour gérer la mémoire.

Les instances de l'interface MemoryManagerMXBean sont obtenues en invoquant la méthode getMemoryManagerMXBeans() de la fabrique ManagementFactory.

L'ObjectName d'une instance de cet MXBean est de la forme : java.lang:type=MemoryManager, name=xxx

Cet MXBean permet d'obtenir plusieurs informations :

- le nom du gestionnaire
- les noms des zones de la mémoire gérées par le gestionnaire

Exemple :

```
package
com.jmdoudoux.tests.jmx.mxbeans;
import
java.lang.management.ManagementFactory;
import
java.lang.management.MemoryManagerMXBean;
public class
TestMemoryManagerMXB {
  public static void main(String[] args) {
```

```
    for (MemoryManagerMXBean mmBean :
ManagementFactory
        .getMemoryManagerMXBeans()) {
      System.out.printf("Memory manager
name: %s\n", mmBean.getName());
      System.out.printf("  isValid : %b\n", mmBean.isValid());
      for (String pool : mmBean.getMemoryPoolNames())
{
        System.out.printf("  Memory pool name : %s\n", pool);
      }
      System.out.println();
    }
  }
}
```

Résultat :

```
Memory manager name: CodeCacheManager
  isValid : true
  Memory pool name : Code Cache

Memory manager name: Copy
  isValid : true
  Memory pool name : Eden Space
  Memory pool name : Survivor Space

Memory manager name: MarkSweepCompact
  isValid : true
  Memory pool name : Eden Space
  Memory pool name : Survivor Space
  Memory pool name : Tenured Gen
  Memory pool name : Perm Gen
  Memory pool name : Perm Gen [shared-ro]
  Memory pool name : Perm Gen [shared-rw]
```

31.15.5. L'interface MemoryMXBean

Cet MXBean permet de surveiller et de gérer la mémoire de la JVM.

Une instance de l'interface MemoryMXBean est obtenue en invoquant la méthode getMemoryMXBean() de la fabrique ManagementFactory.

L'ObjectName pour l'unique instance de cet MXBean est : java.lang:type=Memory

Cet MXBean permet d'obtenir plusieurs informations :

- la quantité de mémoire utilisée dans le tas de la JVM
- la quantité de mémoire utilisée hors du tas dans la JVM
- une estimation du nombre d'objets qui sont en attente de finalisation

Il permet de aussi de demander une exécution du ramasse miette en invoquant sa méthode gc() et d'activer ou non les traces relatives à la gestion de la mémoire sur la sortie standard grâce à la méthode setVerbose().

Exemple :

```
package
com.jmdoudoux.tests.jmx.mxbeans;
import
java.lang.management.ManagementFactory;
import
java.lang.management.MemoryMXBean;
import
java.lang.management.MemoryUsage;
public class TestMemoryMXB
{
```

```
  public static void main(String[] args) {
    MemoryMXBean mbean =
ManagementFactory.getMemoryMXBean();

    System.out.printf("Heap Memory Usage
:\n%s \n", afficherMemoire(mbean.getHeapMemoryUsage()));
    System.out.printf("Non Heap Memory
Usage :\n%s \n", afficherMemoire(mbean.getNonHeapMemoryUsage()));
    System.out.printf("Object pending
finalization : %d\n", mbean.getObjectPendingFinalizationCount() );
    System.out.printf("isVerbose :
%b\n", mbean.isVerbose() );
    System.out.println("Demande
d'execution du ramasse miette");
    mbean.gc();
  }
  public static String
afficherMemoire(MemoryUsage mu) {
    StringBuilder sb= new StringBuilder();
    sb.append("  init = "+mu.getInit()+"\n");
    sb.append("  used = "+mu.getUsed()+"\n");
    sb.append("  commited =
"+mu.getCommitted()+"\n");
    sb.append("  max = "+mu.getMax()+"\n");
    return sb.toString();
  }
}
```

Résultat :

```
Heap Memory Usage :
  init = 0
  used = 223848
  commited = 5177344
  max = 66650112

Non Heap Memory Usage :
  init = 33718272
  used = 13066720
  commited = 34078720
  max = 121634816

Object pending finalization : 0
isVerbose : false
Demande d'execution du ramasse miette
```

La classe MemoryUsage encapsule des données sur l'occupation de la mémoire.

Le MemoryMXBean peut émettre des notifications de deux types :

- MEMORY_THRESHOLD_EXCEED : émise lorsque la quantité de mémoire occupée par un espace mémoire de la JVM a atteint un certain seuil
- MEMORY_COLLECTION_THRESHOLD_EXCEED : émise lorsque la quantité de mémoire occupée par un espace mémoire de la JVM a atteint un certain seuil après l'exécution du ramasse miette

Exemple :

```
package com.jmdoudoux.test.jmx.mxbeans;
import
java.lang.management.ManagementFactory;
import
java.lang.management.MemoryMXBean;
import
java.lang.management.MemoryNotificationInfo;
import
java.lang.management.MemoryPoolMXBean;
import
java.lang.management.MemoryUsage;
import java.util.ArrayList;
```

```
import
javax.management.Notification;
import
javax.management.NotificationEmitter;
import
javax.management.openmbean.CompositeData;
public class
TestMemoryMXBNotif implements
    javax.management.NotificationListener {
  private static final int UN_MEGA_OCTET = 1024
* 1024;
  private static boolean stop = false;
  /**
   * Gestionnaire de traitement de
notifications
   */
  public void handleNotification(Notification
notif, Object handback) {
    System.out.println("\nReception d'une
notification");
    System.out.println("Type : " +
notif.getType());
    System.out.println("Message :  " + notif.getMessage());
    System.out.println("Source objectname
: " + notif.getSource());
    CompositeData cd = (CompositeData)
notif.getUserData();
    MemoryNotificationInfo
memInfo = MemoryNotificationInfo.from( cd );
    System.out.println( "PoolName : "
+ memInfo.getPoolName() );
    MemoryUsage memoryUsage =
memInfo.getUsage();
    System.out.println("\nEtat de la
mémoire");
    System.out.println("  init : " + memoryUsage.getInit());
    System.out.println("  used : " + memoryUsage.getUsed());
    System.out.println("  committed : " +
memoryUsage.getCommitted());
    System.out.println("  max : " + memoryUsage.getMax());
    // arret des traitements
    stop = true;
  }
  public static void main(String[] args) throws
InterruptedException {
    // définition du seuil d'utilisation de la
old generation
    for (MemoryPoolMXBean mpbean :
ManagementFactory.getMemoryPoolMXBeans()) {
      if (mpbean.getName().equals("Tenured
Gen")) {
        if (mpbean.isUsageThresholdSupported())
{
          mpbean.setUsageThreshold(4 *
UN_MEGA_OCTET);
        }
        break;
      }
    }
    // abonnement du listerner auprès du MXBean
    MemoryMXBean mbean = ManagementFactory.getMemoryMXBean();
    ((NotificationEmitter) mbean)
        .addNotificationListener(new
TestMemoryMXBNotif(), null, null);
    // remplissage de la mémoire
    ArrayList<byte[]> list = new
ArrayList<byte[]>();
    int i = 0;
    while (!stop) {
      System.out.printf("iteration %d
\n", ++i);
      list.add(new byte[UN_MEGA_OCTET]);
    }
  }
```

```
}
```

Résultat :

```
iteration 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
23 24 25 26
Reception d'une notification
Type : java.management.memory.threshold.exceeded
Message :  Memory usage
exceeds usage threshold
Source objectname : java.lang:type=Memory
PoolName : Tenured Gen
Etat de la mémoire
  init : 4194304
  used : 4348720
  committed : 5500928
  max : 61997056
```

Remarque : vu le fonctionnement des différents algorithmes utilisés par le ramasse miette, ces notifications ne doivent pas être utilisées pour détecter un manque de mémoire.

31.15.6. L'interface MemoryPoolMXBean

Cet MXBean permet de surveiller les espaces de mémoire de la JVM. Une JVM a plusieurs instances de cet MXBean, une pour chaque espace de mémoire utilisé.

Les instances de l'interface MemoryPoolMXBean sont obtenues en invoquant la méthode getMemoryPoolMXBeans() de la fabrique ManagementFactory.

L'ObjectName d'une instance de cet MXBean est de la forme : java.lang:type=MemoryPool, name=xxx

Cet MXBean permet d'obtenir plusieurs informations :

- une estimation de l'utilisation de l'espace de mémoire
- les valeurs records de l'utilisation de l'espace mémoire
- l'utilisation de l'espace de mémoire après la dernière exécution du ramasse miette

Exemple :

```
package
com.jmdoudoux.test.jmx.mxbeans;
import
java.lang.management.ManagementFactory;
import
java.lang.management.MemoryPoolMXBean;
import java.util.List;
public class
TestMemoryPoolMXB {
  public static void main(String[] args) {
    List<MemoryPoolMXBean>
memoryPoolMXBeans = ManagementFactory
        .getMemoryPoolMXBeans();
    for (MemoryPoolMXBean mpBean :
memoryPoolMXBeans) {
      System.out.printf("Memory Pool Name:
%s\n", mpBean.getName());
      System.out.printf("  Type: %s\n",
mpBean.getType().toString());
      System.out.printf("  isValid: %b\n", mpBean.isValid());
      for (String managerName :
mpBean.getMemoryManagerNames()) {
        System.out.printf("  Memory manager name : %s\n",
managerName);
      }
```

```
      System.out.println();
      System.out.printf("  Usage : %s\n",
mpBean.getUsage().toString());
      System.out.printf("  PeakUsage : %s\n",
mpBean.getPeakUsage().toString());
      boolean bUsageThSupported =
mpBean.isUsageThresholdSupported();
      System.out.printf("  UsageThresholdSupport : %b\n",
bUsageThSupported);
      if (bUsageThSupported) {
        System.out.printf("  UsageThreshold : %d\n", mpBean.getUsageThreshold());
        System.out
            .printf("  UsageThresholdCount : %d\n",
mpBean.getUsageThresholdCount());
        System.out.printf("  isUsageThresholdExceeded : %b\n", mpBean
            .isUsageThresholdExceeded());
      }
      System.out.println();
      System.out.printf("  CollectionUsage: %s\n",
mpBean.getCollectionUsage());
      boolean bCollectionUsageThSupported =
mpBean
          .isCollectionUsageThresholdSupported();
      System.out.printf("  CollectionUsageThresholdSupport: %b\n",
                        bCollectionUsageThSupported);

      if (bCollectionUsageThSupported) {
        System.out.printf("  CollectionUsageThreshold: %d\n", mpBean
            .getCollectionUsageThreshold());
        System.out.printf("  CollectionUsageThresholdcount: %d\n",
mpBean
            .getCollectionUsageThresholdCount());
        System.out.printf("  isCollectionUsageThresholdExceeded:
%b\n", mpBean
            .isCollectionUsageThresholdExceeded());
      }
      System.out.println();
    }
  }
}
```

Le MemoryMXBean peut émettre des notifications de deux types :

- une notification émise lorsque la quantité de mémoire occupée par un espace mémoire de la JVM a atteint un certain seuil
- une notification émise lorsque la quantité de mémoire occupée par un espace mémoire de la JVM a atteint un certain seuil après l'exécution du ramasse miette. Cette notification n'est utilisable qu'avec certains algorithme du ramasse miette.

31.15.7. L'interface OperatingSystemMXBean

Cet MXBean permet d'obtenir quelques informations sur le système d'exploitation.

Une instance de l'interface OperatingSystemMXBean est obtenue en invoquant la méthode getOperatingSystemMXBean() de la fabrique ManagementFactory.

L'ObjectName pour l'unique instance de cet MXBean est : java.lang:type=OperatingSystem

Cet MXBean permet d'obtenir plusieurs informations :

- le type de processeur
- le nombre de processeurs
- le nom du système d'exploitation
- la version du système d'exploitation
- une estimation de la charge du système

Exemple :

```
package com.jmdoudoux.tests.jmx.mxbeans;

import java.lang.management.ManagementFactory;
import java.lang.management.OperatingSystemMXBean;

public class TestOperatingSystemMXB {

  public static void main(String[] args) {

    OperatingSystemMXBean osBean = ManagementFactory.getOperatingSystemMXBean();
    System.out.printf("Arch : %s\n", osBean.getArch());
    System.out.printf("Available processeurs : %d\n", osBean.getAvailableProcessors());
    System.out.printf("Name : %s\n", osBean.getName());
    System.out.printf("System Load Average : %f\n", osBean.getSystemLoadAverage());
    System.out.printf("Version : %s\n", osBean.getVersion());
  }
}
```

Résultat :

```
Arch : x86
Available processeurs : 2
Name : Windows Vista
System Load Average : -1,000000
Version : 6.0
```

Remarque : il est possible d'obtenir la plupart de ces informations soit via des propriétés de la JVM soit via une API

Si la charge système est négative c'est que cette valeur n'est pas disponible.

31.15.8. L'interface RuntimeMXBean

Cet MXBean permet d'obtenir des informations sur la machine virtuelle.

Une instance de l'interface RuntimeMXBean est obtenue en invoquant la méthode getRuntimeMXBean() de la fabrique ManagementFactory.

L'ObjectName pour l'unique instance de cet MXBean est : java.lang:type=Runtime

Cet MXBean permet d'obtenir plusieurs informations sur la JVM.

Exemple :

```
package com.jmdoudoux.tests.jmx.mxbeans;

import java.lang.management.ManagementFactory;
import java.lang.management.RuntimeMXBean;
import java.util.List;
import java.util.Map;

public class TestRuntimeMXB {

  public static void main(String[] args) {

    RuntimeMXBean rBean = ManagementFactory.getRuntimeMXBean();

    System.out.printf("Boot classpath : %s\n", rBean.getBootClassPath());
    System.out.printf("classpath : %s\n", rBean.getClassPath());
    System.out.println("Input argument :");
    List<String> arguments = rBean.getInputArguments();
    for (String arg : arguments) {
      System.out.println("  " + arg);
    }
```

```
    System.out.printf("Library path : %s\n", rBean.getLibraryPath());
    System.out.printf("Management spec version : %s\n", rBean
        .getManagementSpecVersion());
    System.out.printf("Name : %s\n", rBean.getName());
    System.out.printf("Spec name : %s\n", rBean.getSpecName());
    System.out.printf("Vendor : %s\n", rBean.getSpecVendor());
    System.out.printf("Spec version : %s\n", rBean.getSpecVersion());
    System.out.printf("StartTime : %d ms\n", rBean.getStartTime());
    System.out.println("System properties : %s\n");
    Map<String, String> props = rBean.getSystemProperties();
    for (String cle : props.keySet()) {
      System.out.println("  " + cle + " = " + props.get(cle));
    }
    System.out.printf("UpTime : %d ms\n", rBean.getUptime());
    System.out.printf("VmName : %s\n", rBean.getVmName());
    System.out.printf("VmVendor : %s\n", rBean.getVmVendor());
    System.out.printf("VmVersion : %s\n", rBean.getVmVersion());
    System.out.printf("isBootClassPathSupported : %b\n", rBean
        .isBootClassPathSupported());

  }
}
```

Remarque : il est possible d'obtenir la plupart de ces informations soit via des propriétés de la JVM soit via une API

31.15.9. L'interface ThreadMXBean

Cet MXBean permet d'obtenir des informations sur les threads de la JVM.

Une instance de l'interface ThreadMXBean est obtenue en invoquant la méthode getThreadMXBean() de la fabrique ManagementFactory.

L'ObjectName pour l'unique instance de cet MXBean est : java.lang:type=Threading

Cet MXBean permet d'obtenir de nombreuses informations sur les threads de la JVM.

Exemple :

```
package com.jmdoudoux.tests.jmx.mxbeans;

import java.lang.management.ManagementFactory;
import java.lang.management.ThreadInfo;
import java.lang.management.ThreadMXBean;

public class TestThreadMXB {

  public static void main(String[] args) {

    Thread monThread = new Thread(new Runnable() {

      @Override
      public void run() {
        try {
          Thread.sleep(50);
        } catch (InterruptedException e) {
        }
      }

    });

    monThread.setName("Mon Thread");
    monThread.start();

    ThreadMXBean tBean = ManagementFactory.getThreadMXBean();

    System.out.printf("Current thead cpu time : %d\n", tBean
```

```
        .getCurrentThreadCpuTime());
    System.out.printf("Current thread user time : %d\n", tBean
        .getCurrentThreadUserTime());
    System.out.printf("Daemon thread count : %d\n", tBean
        .getDaemonThreadCount());
    System.out.printf("Peak thread count : %d\n", tBean.getPeakThreadCount());
    System.out.printf("Thread count : %d\n", tBean.getThreadCount());
    System.out.printf("Total Started Thread count : %d\n", tBean
        .getTotalStartedThreadCount());
    System.out.println("Liste des threads");
    ThreadInfo[] threads = tBean.dumpAllThreads(false, false);
    for (ThreadInfo ti : threads) {
      System.out.printf("Thead id  : %d\n", ti.getThreadId());
      System.out.printf("  Name : %s\n", ti.getThreadName());
      System.out.printf("  State : %s\n", ti.getThreadState());
      System.out.println("  Stack : ");
      StackTraceElement[] ste = ti.getStackTrace();
      for (StackTraceElement elt : ste) {
        System.out.printf("    %s.%s() - %d\n", elt.getClassName(), elt
            .getMethodName(), elt.getLineNumber());
      }
    }
    monThread.interrupt();
  }
}
```

Résultat :

```
Current thead cpu time : 62400400
Current thread user time : 62400400
Daemon thread count : 4
Peak thread count : 6
Thread count : 6
Total Started Thread count : 6
Liste des threads
Thead id  : 8
  Name : Mon Thread
  State : TIMED_WAITING
  Stack :
    java.lang.Thread.sleep() - -2
    com.jmdoudoux.tests.jmx.mxbeans.TestThreadMXB$1.run() - 17
    java.lang.Thread.run() - 619
Thead id  : 5
  Name : Attach Listener
  State : RUNNABLE
  Stack :
Thead id  : 4
  Name : Signal Dispatcher
  State : RUNNABLE
  Stack :
Thead id  : 3
  Name : Finalizer
  State : WAITING
  Stack :
    java.lang.Object.wait() - -2
    java.lang.ref.ReferenceQueue.remove() - 116
    java.lang.ref.ReferenceQueue.remove() - 132
    java.lang.ref.Finalizer$FinalizerThread.run() - 159
Thead id  : 2
  Name : Reference Handler
  State : WAITING
  Stack :
    java.lang.Object.wait() - -2
    java.lang.Object.wait() - 485
    java.lang.ref.Reference$ReferenceHandler.run() - 116
Thead id  : 1
  Name : main
  State : RUNNABLE
  Stack :
    sun.management.ThreadImpl.dumpThreads0() - -2
    sun.management.ThreadImpl.dumpAllThreads() - 374
    com.jmdoudoux.tests.jmx.mxbeans.TestThreadMXB.main() - 36
```

31.15.10. La sécurisation des accès à l'agent

La suite de cette section sera développée dans une version future de ce document

31.16. Des recommandations pour l'utilisation de JMX

Il faut être vigilant sur les ObjectNames associés aux MBeans en définissant des conventions de nommage notamment pour permettre leur organisation d'une façon hiérarchique dans les clients JMX. Il peut par exemple être intéressant d'utiliser un ou plusieurs attributs pour structurer cette hiérarchie et toujours avoir un attribut qui précise le type du MBean.

Pour améliorer la portabilité, il est préférable d'utiliser les Open MBeans ou d'utiliser les Open Types pour les structures de données utilisées par les MBeans plutôt que d'utiliser des types personnalisés.

Il est préférable d'utiliser les MBeans standards lorsque cela est possible car ils sont faciles à écrire et à maintenir.

Le monitoring implique généralement la récupération et l'exploitation de différentes données : états, métriques, statistiques, ... Pour l'exploitation de ces données, leur affichage peut avoir des conséquences sur les fonctionnalités des MBeans.

Pour avoir une vision d'ensemble au niveau d'un domaine regroupant plusieurs applications et au niveau de tout ou partie du système d'informations, il faut agréger les données collectées de toutes les JVM.

Typiquement un client de monitoring interroge périodiquement le système pour afficher des données fraiches. Ce procédé peut être consommateur en terme de ressources (CPU, bande passante, ...). Les notifications peuvent être une solution dans certains cas mais mal utilisé cela peut être encore pire. Elles sont toujours plus intéressantes lorsqu'une donnée évolue peu : dans ce cas, il est préférable d'émettre une notification à chaque changement de valeur plutôt que de périodiquement récupérer une valeur qui sera fréquemment identique.

31.17. Des ressources

La page principale de la technologie JMX

http://java.sun.com/jmx

La documentation de JMX

http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement/docs.jsp

La documentation de l'API Jmx

http://java.sun.com/j2se/1.5.0/docs/guide/jmx/

Un article sur l'utilisation de JConsole

http://java.sun.com/developer/technicalArticles/J2SE/jconsole.html

JMX best practice

http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement/best-practices.jsp

mx4j est une implémentation open source de JMX

http://mx4j.sourceforge.net/

JbossMX est un implémentation open source de JMX

http://www.jboss.org/community/wiki/JBossMX

MC4J est un client JMX open source

http://mc4j.org

JManage est une application de gestion open source

http://www.jmanage.org

Partie 4 : l'utilisation de documents XML

Cette partie traite de l'utilisation de documents XML avec Java. L'utilisation de documents XML peut se faire au travers de plusieurs API.

Cette partie regroupe plusieurs chapitres :

- Java et XML : présente XML qui est une technologie qui c'est imposée pour les échanges de données et explore les API Java pour utiliser XML
- SAX (Simple API for XML) : présente l'utilisation de l'API SAX avec Java. Cette API utilise des événements pour traiter un document XML
- DOM (Document Object Model) : présente l'utilisation avec Java de cette spécification du W3C pour proposer une API qui permet de modéliser, de parcourir et de manipuler un document XML
- XSLT (Extensible Stylesheet Language Transformations) : présente l'utilisation avec Java de cette recommandation du W3C pour transformer des documents XML
- Les modèles de document : présente quelques API open source spécifiques à Java pour traiter un document XML : JDom et Dom4J
- JAXB (Java Architecture for XML Binding) : détaille l'utilisation de cette spécification qui permet de faire correspondre un document XML à un ensemble de classes et vice et versa.
- StAX (Streaming Api for XML) : détaille l'utilisation de cette API qui permet de traiter un document XML de façon simple en consommant peut de mémoire tout en permettant de garder le contrôle sur les opérations d'analyse ou d'écriture

32. Java et XML

L'utilisation ensemble de Java et XML est facilitée par le fait qu'ils ont plusieurs points communs :

- indépendance de toute plateforme
- conçu pour être utilisé sur un réseau
- prise en charge de la norme Unicode

Ce chapitre contient plusieurs sections :

- La présentation de XML
- Les règles pour formater un document XML
- La DTD (Document Type Definition)
- Les parseurs
- La génération de données au format XML
- JAXP : Java API for XML Parsing
- Jaxen

32.1. La présentation de XML

XML est l'acronyme de «eXtensible Markup Language».

XML permet d'échanger des données entres applications hétérogènes car il permet de modéliser et de stocker des données de façon portable.

XML est extensible dans la mesure où il n'utilise pas de tags prédéfinis comme HTML et il permet de définir de nouvelles balises : c'est un métalangage.

Le format HTML est utilisé pour formater et afficher les données qu'il contient : il est destiné à structurer, formater et échanger des documents d'une façon la plus standard possible..

XML est utilisé pour modéliser et stocker des données. Il ne permet pas à lui seul d'afficher les données qu'il contient.

Pourtant, XML et HTML sont tous les deux des dérivés d'un langage nommé SGML (Standard Generalized Markup Language). La création d'XML est liée à la complexité de SGML. D'ailleurs, un fichier XML avec sa DTD correspondante peut être traité par un processeur SGML.

XML et Java ont en commun la portabilité réalisée grâce à une indépendance vis à vis du système et de leur environnement.

32.2. Les règles pour formater un document XML

Un certain nombre de règles doivent être respectées pour définir un document XML valide et «bien formé». Pour pouvoir être analysé, un document XML doit avoir une syntaxe correcte. Les principales règles sont :

- le document doit contenir au moins une balise
- chaque balise d'ouverture (exemple <tag>) doit posséder une balise de fermeture (exemple </tag>). Si le tag est vide, c'est à dire qu'il ne possède aucune données (exemple <tag></tag>), un tag abrégé peut être utilisé (exemple correspondant : <tag/>)
- les balises ne peuvent pas être intercalées (exemple <liste><element></liste></element> n'est pas autorisé)
- toutes les balises du document doivent obligatoirement être contenues entre une balise d'ouverture et de fermeture unique dans le document nommée élément racine
- les valeurs des attributs doivent obligatoirement être encadrées avec des quotes simples ou doubles
- les balises sont sensibles à la casse
- Les balises peuvent contenir des attributs même les balises vides
- les données incluses entre les balises ne doivent pas contenir de caractères < et & : il faut utiliser respectivement < ; et & ;
- La première ligne du document devrait normalement correspondre à la déclaration de document XML : le prologue.

32.3. La DTD (Document Type Definition)

Les balises d'un document XML sont libres. Pour pouvoir valider si le document est correct, il faut définir un document nommé DTD qui est optionnel. Sans sa présence, le document ne peut être validé : on peut simplement vérifier que la syntaxe du document est correcte.

Une DTD est un document qui contient la grammaire définissant le document XML. Elle précise notamment les balises autorisées et comment elles s'imbriquent.

La DTD peut être incluse dans l'en tête du document XML ou être mise dans un fichier indépendant. Dans ce cas, la directive < !DOCTYPE> dans le document XML permet de préciser le fichier qui contient la DTD.

Il est possible d'utiliser une DTD publique ou de définir sa propre DTD si aucune ne correspond à ses besoins.

Pour être valide, un document XML doit avoir une syntaxe correcte et correspondre à la DTD.

32.4. Les parseurs

Il existe plusieurs types de parseur. Les deux plus répandus sont ceux qui utilisent un arbre pour représenter et exploiter le document et ceux qui utilisent des événements. Le parseur peut en plus permettre de valider le document XML.

Ceux qui utilisent un arbre permettent de le parcourir pour obtenir les données et modifier le document.
Ceux qui utilisent des événements associent à des événements particuliers des méthodes pour traiter le document.

SAX (Simple API for XML) est une API libre créée par David Megginson qui utilise les événements pour analyser et exploiter les documents au format XML.

Les parseurs qui produisent des objets composant une arborescence pour représenter le document XML utilisent le modèle DOM (Document Object Model) défini par les recommandations du W3C.

Le choix d'utiliser SAX ou DOM doit tenir compte de leurs points forts et de leurs faiblesses :

	les avantages	les inconvénients
DOM	parcours libre de l'arbre possibilité de modifier la structure et le contenu de l'arbre	gourmand en mémoire doit traiter tout le document avant d'exploiter les résultats
SAX	peu gourmand en ressources mémoire rapide	traite les données séquentiellement un peu plus difficile à programmer, il est souvent nécessaire de sauvegarder des

	principes faciles à mettre en oeuvre permet de ne traiter que les données utiles	informations pour les traiter

SAX et DOM ne fournissent que des définitions : ils ne fournissent pas d'implémentation utilisable. L'implémentation est laissée aux différents éditeurs qui fournissent un parseur compatible avec SAX et/ou DOM.L'avantage d'utiliser l'un d'eux est que le code utilisé sera compatible avec les autres : le code nécessaire à l'instanciation du parseur est cependant spécifique à chaque fournisseur.

IBM fourni gratuitement un parseur XML : xml4j. Il est téléchargeable à l'adresse suivante : http://www.alphaworks.ibm.com/tech/xml4j

Le groupe Apache développe Xerces à partir de xml4j : il est possible de télécharger la dernière version à l'URL http://xml.apache.org

Sun a développé un projet dénommé Project X. Ce projet a été repris par le groupe Apache sous le nom de Crimson.

Ces trois projets apportent pour la plupart les mêmes fonctionnalités : ils se distinguent sur des points mineurs : performance, rapidité, facilité d'utilisation etc. ... Ces fonctionnalités évoluent très vite avec les versions de ces parseurs qui se succèdent très rapidement.

Pour les utiliser, il suffit de décompresser le fichier et d'ajouter les fichiers .jar dans la variable définissant le CLASSPATH.

Il existe plusieurs autres parseurs que l'on peut télécharger sur le web.

32.5. La génération de données au format XML

Il existe plusieurs façons de générer des données au format XML :

- coder cette génération à la main en écrivant dans un flux

Exemple :

```
public void service(HttpServletRequest request, HttpServletResponse response)
  throws IOException, ServletException {
  response.setContentType("text/xml");
  PrintWriter out = response.getWriter();

  out.println("<?xml version=\"1.0\"?>");
  out.println("<BIBLIOTHEQUE>");
  out.println(" <LIVRE>");
  out.println(" <TITRE>titre livre 1</TITRE>");
  out.println(" <AUTEUR>auteur 1</AUTEUR>");
  out.println(" <EDITEUR>editeur 1</EDITEUR>");
  out.println(" </LIVRE>");
  out.println(" <LIVRE>");
  out.println(" <TITRE>titre livre 2</TITRE>" );
  out.println(" <AUTEUR>auteur 2</AUTEUR>");
  out.println(" <EDITEUR>editeur 2</EDITEUR>" );
  out.println(" </LIVRE>");
  out.println(" <LIVRE>");
  out.println(" <TITRE>titre livre 3</TITRE>");
  out.println(" <AUTEUR>auteur 3</AUTEUR>");
  out.println(" <EDITEUR>editeur 3</EDITEUR>");
  out.println(" </LIVRE>");
  out.println("</BIBLIOTHEQUE>");
  }
}
```

- utiliser JDOM pour construire le document et le sauvegarder

* utiliser la classe javax.xml.stream.XmlStreamWriter
* utiliser la classe java.beans.XMLEncoder pour sérialiser un bean
* utiliser une API open source comme XStream

La suite de cette section sera développée dans une version future de ce document

32.6. JAXP : Java API for XML Parsing

JAXP est une API développée par Sun qui ne fournit pas une nouvelle méthode pour parser un document XML mais propose une interface commune pour appeler et paramétrer un parseur de façon indépendante de tout fournisseur et normaliser la source XML à traiter. En utilisant un code qui respecte JAXP, il est possible d'utiliser n'importe quel parseur qui répond à cette API tel que Crimson le parseur de Sun ou Xerces le parseur du groupe Apache.

JAXP supporte pour le moment les parseurs de type SAX et DOM.

	JAXP 1.0	JAXP 1.1
SAX	type 1	type 2
DOM	niveau 1	niveau 2

Par exemple, sans utiliser JAXP, il existe deux méthodes pour instancier un parseur de type SAX :

* créer une instance de la classe de type SaxParser
* utiliser la classe ParserFactory qui demande en paramètre le nom de la classe de type SaxParser

Ces deux possibilités nécessitent une recompilation d'une partie du code lors du changement du parseur.

JAXP propose de fournir le nom de la classe du parseur en paramètre à la JVM sous la forme d'une propriété système. Il n'est ainsi plus nécessaire de procéder à une recompilation mais simplement de mettre jour cette propriété et le CLASSPATH pour qu'il référence les classes du nouveau parseur.

Le parseur de Sun et les principaux parseurs XML en Java implémentent cette API et il est très probable que tous les autres fournisseurs suivent cet exemple.

32.6.1. JAXP 1.1

JAXP version 1.1 contient une documentation au format javadoc, des exemples et trois fichiers jar :

* jaxp.jar : contient l'API JAXP
* crimson.jar : contient le parseur de Sun
* xalan.jar : contient l'outil du groupe apache pour les transformations XSL

L'API JAXP est fournie avec une implémentation de référence de deux parseurs (une de type SAX et une de type DOM) dans le package org.apache.crimson et ses sous packages.

JAXP se compose de plusieurs packages :

* javax.xml.parsers
* javax.xml.transform
* org.w3c.dom
* org.xml.sax

JAXP définit deux exceptions particulières :

- FactoryConfigurationError est levée si la classe du parseur précisée dans la variable System ne peut être instanciée
- ParserConfigurationException est levée lorsque les options précisées dans la factory ne sont pas supportées par le parseur

32.6.2. L'utilisation de JAXP avec un parseur de type SAX

L'API JAXP fournit la classe abstraite SAXParserFactory qui propose une méthode pour récupérer une instance d'un parseur de type SAX grâce à une de ces méthodes. Une classe fille de la classe SAXParserFactory permet d'être instanciée.

La propriété système javax.xml.parsers.SAXParserFActory permet de préciser la classe fille qui hérite de la classe SAXParserFactory et qui sera instanciée.

Remarque : cette classe n'est pas thread safe.

La méthode statique newInstance() permet d'obtenir une instance de la classe SAXParserFactory : elle peut lever une exception de type FactoryConfigurationError.

Avant d'obtenir une instance du parseur, il est possible de fournir quelques paramètres à la Factory pour lui permettre de configurer le parseur.

La méthode newSAXParser() permet d'obtenir une instance du parseur de type SAXParser : peut lever une exception de type ParserConfigurationException.

Les principales méthodes sont :

Méthode	Rôle
boolean isNamespaceAware()	indique si la factory est configurée pour instancier des parseurs qui prennent en charge les espaces de noms
boolean isValidating()	indique si la factory est configurée pour instancier des parseurs qui valide le document XML lors de son traitement
static SAXParserFactory newInstance()	permet d'obtenir une instance de la factory
SAXParser newSAXParser()	permet d'obtenir une nouvelle instance du parseur de type SAX configuré avec les options fournie à la factory
setNamespaceAware(boolean)	configure la factory pour instancier un parseur qui prend en charge les espaces de noms ou non selon le paramètre fourni
setValidating(boolean)	configure la factory pour instancier un parseur qui valide le document XML lors de son traitement ou non selon le paramètre fourni

Exemple :

```
SAXParserFActory factory = SAXParserFactory.newInstance();
SAXParser parser = factory.newSAXParser();
parser.parse(new File(args[0), new handler());
```

32.7. Jaxen

Jaxen est un moteur Xpath qui permet de retrouver des informations grâce à Xpath dans un document XML de type dom4j ou Jdom.

C'est un projet open source qui a été intégré dans dom4j pour permettre le support de Xpath dans ce framework.

33. SAX (Simple API for XML)

33.1. L'utilisation de SAX

SAX est l'acronyme de Simple API for XML. Cette API a été développée par David Megginson.

Ce type de parseur utilise des événements pour piloter le traitement d'un fichier XML. Un objet (nommé handler en anglais) doit implémenter des méthodes particulières définies dans une interface de l'API pour fournir les traitements à réaliser : selon les événements, le parseur appelle ces méthodes.

Les dernières informations concernant cette API sont disponible à l'URL : www.megginson.com/SAX/index.html

Les classes de l'API SAX sont regroupées dans le package org.xml.sax

33.1.1. L'utilisation de SAX de type 1

SAX type 1 est composé de deux packages :

- org.xml.sax :
- org.xml.sax.helpers :

SAX définit plusieurs classes et interfaces :

- les interfaces implémentées par le parseur : Parser, AttributeList et Locator
- les interfaces implémentées par le handler : DocumentHandler, ErrorHandler, DTDHandler et EntityHandler
- les classes de SAX :
- des utilitaires rassemblés dans le package org.xml.sax.helpers notamment la classe ParserFactory

Les exemples de cette section utilisent la version 2.0.15 du parseur xml4j d'IBM.

Pour parser un document XML avec un parseur XML SAX de type 1, il faut suivre les étapes suivantes :

- créer une classe qui implémente l'interface DocumentHandler ou hérite de la classe org.xml.sax.HandlerBase et qui se charge de répondre aux différents événements émis par le parseur
- créer une instance du parseur en utilisant la méthode makeParser() de la classe ParserFactory.
- associer le handler au parseur grâce à la méthode setDocumentHandler()
- exécuter la méthode parse() du parseur

Exemple : avec XML4J

```
import org.xml.sax.*;
import org.xml.sax.helpers.ParserFactory;
import com.ibm.xml.parsers.*;
import java.io.*;

public class MessageXML {
  static final String DONNEES_XML =
  "<?xml version=\"1.0\"?>\n"
  +"<BIBLIOTHEQUE\n>"
```

```
  +" <LIVRE>\n"
  +" <TITRE>titre livre 1</TITRE>\n"
  +" <AUTEUR>auteur 1</AUTEUR>\n"
  +" <EDITEUR>editeur 1</EDITEUR>\n"
  +" </LIVRE>\n"
  +" <LIVRE>\n"
  +" <TITRE>titre livre 2</TITRE>\n"
  +" <AUTEUR>auteur 2</AUTEUR>\n"
  +" <EDITEUR>editeur 2</EDITEUR>\n"
  +" </LIVRE>\n"
  +" <LIVRE>\n"
  +" <TITRE>titre livre 3</TITRE>\n"
  +" <AUTEUR>auteur 3</AUTEUR>\n"
  +" <EDITEUR>editeur 3</EDITEUR>\n"
  +" </LIVRE>\n"
  +"</BIBLIOTHEQUE>\n";

  static final String CLASSE_PARSER = "com.ibm.xml.parsers.SAXParser";

  /**
   * Lance l'application.
   * @param args un tableau d'arguments de ligne de commande
   */
  public static void main(java.lang.String[] args) {

    MessageXML m = new MessageXML();
    m.parse();

    System.exit(0);
  }

  public MessageXML() {
    super();
  }

  public void parse() {
    TestXMLHandler handler = new TestXMLHandler();

    System.out.println("Lancement du parseur");

    try {
      Parser parser = ParserFactory.makeParser(CLASSE_PARSER);

      parser.setDocumentHandler(handler);
      parser.setErrorHandler((ErrorHandler) handler);

      parser.parse(new InputSource(new StringReader(DONNEES_XML)));

    } catch (Exception e) {
      System.out.println("Exception capturée : ");
      e.printStackTrace(System.out);
      return;
    }
  }
}
```

Il faut ensuite créer la classe du handler.

Exemple :

```
import java.util.*;

/**
 * Classe utilisee pour gérer les evenement emis par SAX lors du traitement du fichier XML
 */
public class TestXMLHandler extends org.xml.sax.HandlerBase {
  public TestXMLHandler() {
    super();
  }
```

```
  /**
   * Actions à réaliser sur les données
   */
  public void characters(char[] caracteres, int debut, int longueur) {
    String donnees = new String(caracteres, debut, longueur);
    System.out.println("   valeur = *" + donnees + "*");
  }

  /**
   * Actions à réaliser lors de la fin du document XML.
   */
  public void endDocument() {
    System.out.println("Fin du document");
  }

  /**
   * Actions à réaliser lors de la détection de la fin d'un element.
   */
  public void endElement(String name) {
    System.out.println("Fin tag " + name);
  }

  /**
   * Actions à réaliser au début du document.
   */
  public void startDocument() {
    System.out.println("Debut du document");
  }

  /**
   * Actions a réaliser lors de la detection d'un nouvel element.
   */
  public void startElement(String name, org.xml.sax.AttributeList atts) {
    System.out.println("debut tag : " + name);
  }
}
```

Résultat :

```
Lancement du parser
Debut du document
debut tag : BIBLIOTHEQUE
   valeur = *
  *
debut tag : LIVRE
   valeur = *
    *
debut tag : TITRE
   valeur = *titre livre 1*
Fin tag TITRE
   valeur = *
    *
debut tag : AUTEUR
   valeur = *auteur 1*
Fin tag AUTEUR
   valeur = *
    *
debut tag : EDITEUR
   valeur = *editeur 1*
Fin tag EDITEUR
   valeur = *
  *
Fin tag LIVRE
   valeur = *
  *
debut tag : LIVRE
   valeur = *
    *
debut tag : TITRE
   valeur = *titre livre 2*
Fin tag TITRE
   valeur = *
```

```
    *
debut tag : AUTEUR
   valeur = *auteur 2*
Fin tag AUTEUR
   valeur = *
    *
debut tag : EDITEUR
   valeur = *editeur 2*
Fin tag EDITEUR
   valeur = *
  *
Fin tag LIVRE
   valeur = *
  *
debut tag : LIVRE
   valeur = *
    *
debut tag : TITRE
   valeur = *titre livre 3*
Fin tag TITRE
   valeur = *
    *
debut tag : AUTEUR
   valeur = *auteur 3*
Fin tag AUTEUR
   valeur = *
    *
debut tag : EDITEUR
   valeur = *editeur 3*
Fin tag EDITEUR
   valeur = *
  *
Fin tag LIVRE
   valeur = *
*
Fin tag BIBLIOTHEQUE
Fin du document
```

Un parseur SAX peut créer plusieurs types d'événements dont les principales méthodes pour y répondre sont :

Evénement	Rôle
startElement()	cette méthode est appelée lors de la détection d'un tag de début
endElement()	cette méthode est appelée lors de la détection d'un tag de fin
characters()	cette méthode est appelée lors de la détection de données entre deux tags
startDocument()	cette méthode est appelée lors du début du traitement du document XML
endDocument()	cette méthode est appelée lors de la fin du traitement du document XML

La classe handler doit redéfinir certaines de ces méthodes selon les besoins des traitements.

En règle générale :

- il faut sauvegarder dans une variable le tag courant dans la méthode startElement()
- traiter les données en fonction du tag courant dans la méthode characters()

La sauvegarde du tag courant est obligatoire car la méthode characters() ne contient pas dans ses paramètres le nom du tag correspondant aux données.

Si les données contenues dans le document XML contiennent plusieurs occurrences qu'il faut gérer avec une collection qui contiendra des objets encapsulant les données, il faut :

- gérer la création d'un objet dans la méthode startElement() lors de la rencontre du tag de début d'un nouvel élément de la liste

* alimenter les attributs de l'objet avec les données de chaque tag utile dans la méthode characters()
* gérer l'ajout de l'objet à la collection dans la méthode endElement() lors de la rencontre du tag de fin d'élément de la liste

La méthode characters() est appelée lors de la détection de données entre un tag de début et un tag de fin mais aussi entre un tag de fin et le tag début suivant lorsqu'il y a des caractères entre les deux. Ces caractères ne sont pas des données mais des espaces, des tabulations, des retour chariots et certains caractères non visibles.

Pour éviter de traiter les données de ces événements, il y a plusieurs solutions :

* supprimer tous les caractères entre les tags : tous les tags et les données sont rassemblés sur une seule et unique ligne. L'inconvénient de cette méthode est que le message est difficilement lisible par un être humain.
* une autre méthode consiste à remettre à vide la donnée qui contient le tag courant (alimentée dans la méthode startElement()) dans la méthode endElement(). Il suffit alors d'effectuer les traitements dans la méthode characters() uniquement si le tag courant est différent de vide

Exemple :

```
import java.util.*;

/**
 * Classe utilisee pour gérer les evenement emis par SAX lors du traitement du fichier XML
 */
public class TestXMLHandler extends org.xml.sax.HandlerBase {
  private String tagCourant = "";
  public TestXMLHandler() {
    super();
  }

  /**
   * Actions à réaliser sur les données
   */
  public void characters(char[] caracteres, int debut, int longueur) {
    String donnees = new String(caracteres, debut, longueur);
    if (!tagCourant.equals("") {
      System.out.println("  Element " + tagCourant
        + ", valeur = *" + donnees + "*");
    }
  }

  /**
   * Actions à réaliser lors de la fin du document XML.
   */
  public void endDocument() {
    System.out.println("Fin du document");
  }

  /**
   * Actions à réaliser lors de la détection de la fin d'un element.
   */
  public void endElement(String name) {
    tagCourant = "";
    System.out.println("Fin tag " + name);
  }

  /**
   * Actions à réaliser au début du document.
   */
  public void startDocument() {
    System.out.println("Debut du document");
  }

  /**
   * Actions a réaliser lors de la detection d'un nouvel element.
   */
  public void startElement(String name, org.xml.sax.AttributeList atts) {
    tagCourant = name;
    System.out.println("debut tag : " + name);
```

```
    }
}
```

Résultat :

```
Lancement du parser
Debut du document
debut tag : BIBLIOTHEQUE
   Element BIBLIOTHEQUE, valeur = *
  *
debut tag : LIVRE
   Element LIVRE, valeur = *
    *
debut tag : TITRE
   Element TITRE, valeur = *titre livre 1*
Fin tag TITRE
debut tag : AUTEUR
   Element AUTEUR, valeur = *auteur 1*
Fin tag AUTEUR
debut tag : EDITEUR
   Element EDITEUR, valeur = *editeur 1*
Fin tag EDITEUR
Fin tag LIVRE
debut tag : LIVRE
   Element LIVRE, valeur = *
    *
debut tag : TITRE
   Element TITRE, valeur = *titre livre 2*
Fin tag TITRE
debut tag : AUTEUR
   Element AUTEUR, valeur = *auteur 2*
Fin tag AUTEUR
debut tag : EDITEUR
   Element EDITEUR, valeur = *editeur 2*
Fin tag EDITEUR
Fin tag LIVRE
debut tag : LIVRE
   Element LIVRE, valeur = *
    *
debut tag : TITRE
   Element TITRE, valeur = *titre livre 3*
Fin tag TITRE
debut tag : AUTEUR
   Element AUTEUR, valeur = *auteur 3*
Fin tag AUTEUR
debut tag : EDITEUR
   Element EDITEUR, valeur = *editeur 3*
Fin tag EDITEUR
Fin tag LIVRE
Fin tag BIBLIOTHEQUE
Fin du document
```

- enfin il est possible de vérifier si le premier caractère des données contenues en paramètre de la méthode characters() est un caractère de contrôle ou non grâce à la méthode statique isISOControl() de la classe Character

Exemple :

```
...
/**
 * Actions à réaliser sur les données
 */
public void characters(char[] caracteres, int debut, int longueur) {
        String donnees = new String(caracteres, debut, longueur);

        if (!tagCourant.equals("")) {
                if(!Character.isISOControl(caracteres[debut])) {
                        System.out.println("   Element " + tagCourant
```

```
                  +", valeur = *" + donnees + "*");
                }
        }
}
...
```

Résultat :

```
Lancement du parser
Debut du document
debut tag : BIBLIOTHEQUE
debut tag : LIVRE
debut tag : TITRE
   Element TITRE, valeur = *titre livre 1*
Fin tag TITRE
debut tag : AUTEUR
   Element AUTEUR, valeur = *auteur 1*
Fin tag AUTEUR
debut tag : EDITEUR
   Element EDITEUR, valeur = *editeur 1*
Fin tag EDITEUR
Fin tag LIVRE
debut tag : LIVRE
debut tag : TITRE
   Element TITRE, valeur = *titre livre 2*
Fin tag TITRE
debut tag : AUTEUR
   Element AUTEUR, valeur = *auteur 2*
Fin tag AUTEUR
debut tag : EDITEUR
   Element EDITEUR, valeur = *editeur 2*
Fin tag EDITEUR
Fin tag LIVRE
debut tag : LIVRE
debut tag : TITRE
   Element TITRE, valeur = *titre livre 3*
Fin tag TITRE
debut tag : AUTEUR
   Element AUTEUR, valeur = *auteur 3*
Fin tag AUTEUR
debut tag : EDITEUR
   Element EDITEUR, valeur = *editeur 3*
Fin tag EDITEUR
Fin tag LIVRE
Fin tag BIBLIOTHEQUE
Fin du document
```

SAX définit une exception de type SAXParserException lorsque le parseur détecte une erreur dans le document en cours de traitement. Les méthodes getLineNumber() et getColumnNumber() permettent d'obtenir la ligne et la colonne où l'erreur a été détectée.

Exemple :

```
try {
...
}  catch (SAXParseException e) {
  System.out.println("Erreur lors du traitement du document XML");
  System.out.println(e.getMessage());
  System.out.println("ligne : "+e.getLineNumber());
  System.out.println("colonne : "+e.getColumnNumber());
  }
```

Pour les autres erreurs, SAX définit l'exception SAXException.

33.1.2. L'utilisation de SAX de type 2

SAX de type 2 apporte principalement le support des espaces de noms. Les classes et les interfaces sont toujours définies dans les packages org.xml.sax et ses sous packages.

SAX de type 2 définit quatre interfaces que l'objet handler doit ou peut implémenter :

- ContentHandler : interface qui définit les méthodes appelées lors du traitement du document
- ErrorHandler : interface qui définit les méthodes appelées lors du traitement des warnings et des erreurs
- DTDHandler : interface qui définit les méthodes appelées lors du traitement de la DTD
- EntityResolver

Plusieurs classes et interfaces de SAX de type 1 sont deprecated :

	ancienne entité SAX 1	nouvelle entité SAX 2
Interface	org.xml.sax.Parser	XMLReader
	org.xml.sax.DocumentHandler	ContentHandler
	org.xml.sax.AttributeList	Attributes
Classes	org.xml.sax.helpers.ParserFactory	
	org.xml.sax.HandlerBase	DefaultHandler
	org.xml.sax.helpers.AttributeListImpl	AttributesImpl

Les principes de fonctionnement de SAX 2 sont très proche de SAX 1.

Exemple :

```
import org.xml.sax.*;
import org.xml.sax.helpers.*;

public class TestSAX2
{

  public static void main(String[] args)
  {
    try
    {
      Class c = Class.forName("org.apache.xerces.parsers.SAXParser");
      XMLReader reader = (XMLReader)c.newInstance();
      TestSAX2Handler handler = new TestSAX2Handler();
      reader.setContentHandler(handler);
      reader.parse("test.xml");
    }
    catch(Exception e){System.out.println(e);}
  }

}


class TestSAX2Handler extends DefaultHandler
{
  private String tagCourant = "";

  /**
   * Actions a réaliser lors de la detection d'un nouvel element.
   */
  public void startElement(String nameSpace, String localName,
    String qName, Attributes attr) throws SAXException  {
    tagCourant = localName;
    System.out.println("debut tag : " + localName);
  }

  /**
   * Actions à réaliser lors de la détection de la fin d'un element.
```

```
   */
  public void endElement(String nameSpace, String localName,
    String qName) throws SAXException {
    tagCourant = "";
    System.out.println("Fin tag " + localName);
  }

  /**
   * Actions à réaliser au début du document.
   */
  public void startDocument() {
    System.out.println("Debut du document");
  }

  /**
   * Actions à réaliser lors de la fin du document XML.
   */
  public void endDocument() {
    System.out.println("Fin du document");
  }

  /**
   * Actions à réaliser sur les données
   */
  public void characters(char[] caracteres, int debut,
    int longueur) throws SAXException {
    String donnees = new String(caracteres, debut, longueur);

    if (!tagCourant.equals("")) {
      if(!Character.isISOControl(caracteres[debut])) {
        System.out.println("   Element " + tagCourant +",
          valeur = *" + donnees + "*");
      }
    }
  }
}
```

34. DOM (Document Object Model)

DOM est l'acronyme de Document Object Model. C'est une spécification du W3C pour proposer une API qui permet de modéliser, de parcourir et de manipuler un document XML.

Le principal rôle de DOM est de fournir une représentation mémoire d'un document XML sous la forme d'un arbre d'objets et d'en permettre la manipulation (parcours, recherche et mise à jour)

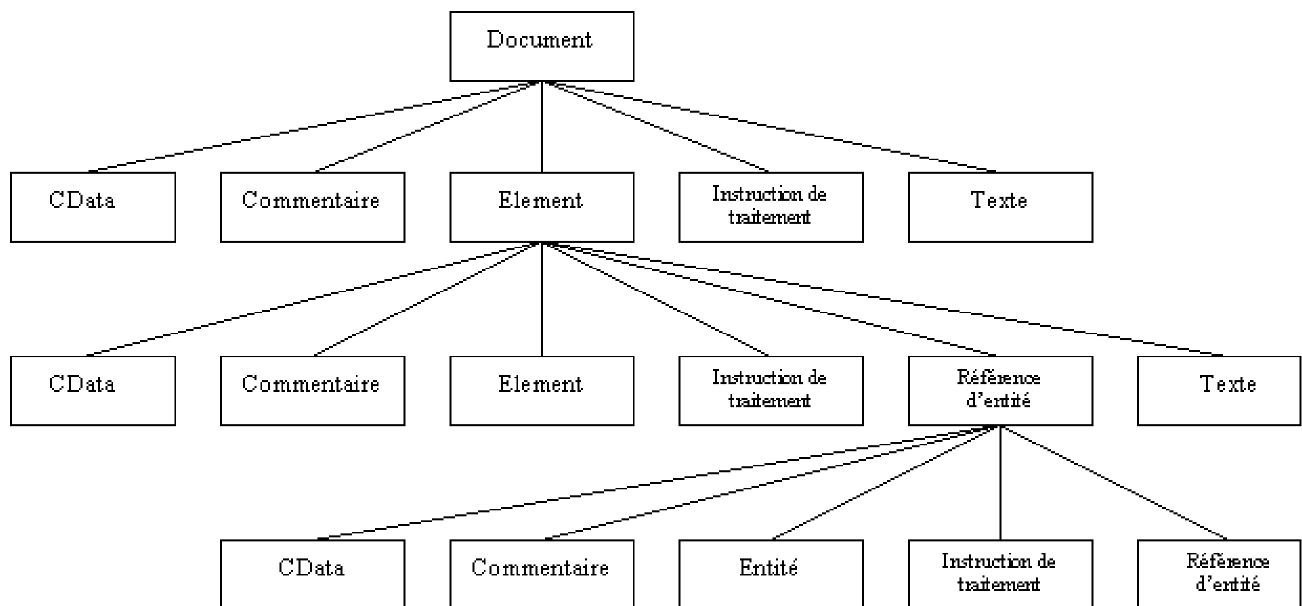
A partir de cette représentation (le modèle), DOM propose de parcourir le document mais aussi de pouvoir le modifier. Ce dernier aspect est l'un des aspects les plus intéressants de DOM.

DOM est défini pour être indépendant du langage dans lequel il sera implémenté. DOM n'est qu'une spécification qui pour être utilisée doit être implémentée par un éditeur tiers.

Il existe deux versions de DOM nommées «niveau» :

- DOM Core Level 1 : cette spécification contient les bases pour manipuler un document XML (document, élément et noeud)
- DOM level 2

Le level 3 est en cours de développement.



Chaque élément qui compose l'arbre possède un type. Selon ce type, l'élément peut avoir certains éléments fils comme le montre le schéma ci dessus :

Le premier élément est le document encapsulé dans l'interface Document

Toutes les classes et interfaces sont regroupées dans le package org.w3c.dom

34.1. Les interfaces du DOM

Chaque type d'entité qui compose l'arbre est défini dans une interface. L'interface de base est l'interface Node. Plusieurs autres interfaces héritent de cette interface.

34.1.1. L'interface Node

Chaque élément de l'arbre est un noeud encapsulé dans l'interface org.w3c.dom.Node ou dans une de ses interfaces filles.

L'interface définit plusieurs méthodes :

Méthode	Rôle
short getNodeType()	Renvoyer le type du noeud
String getNodeName()	Renvoyer le nom du noeud
String getNodeValue()	Renvoyer la valeur du noeud
NamedNodeList getAttributes()	Renvoyer la liste des attributs ou null
void setNodeValue(String)	Mettre à jour la valeur du noeud
boolean hasChildNodes()	Renvoyer un booléen qui indique si le noeud à au moins un noeud fils
Node getFirstChild()	Renvoyer le premier noeud fils du noeud ou null
Node getLastChild()	Renvoyer le dernier noeud fils du noeud ou null
NodeList getChildNodes()	Renvoyer une liste des noeuds fils du noeud ou null
Node getParentNode()	Renvoyer le noeud parent du noeud ou null
Node getPreviousSibling()	Renvoyer le noeud frère précédent
Node getNextSibling()	Renvoyer le noeud frère suivant
Document getOwnerDocument()	Renvoyer le document dans lequel le noeud est inclus
Node insertBefore(Node, Node)	Insèrer le premier noeud fourni en paramètre avant le second noeud
Node replaceNode(Node, Node)	Remplacer le second noeud fourni en paramètre par le premier
Node removeNode(Node)	Supprimer le noeud fourni en paramètre
Node appendChild(Node)	Ajouter le noeud fourni en paramètre aux noeuds enfants du noeud courant
Node clodeNode(boolean)	Renvoyer une copie du noeud. Le booléen fourni en paramètre indique si la copie doit inclure les noeuds enfants

Tous les différents noeuds qui composent l'arbre héritent de cette interface. La méthode getNodeType() permet de connaître le type du noeud. Le type est très important car il permet de savoir ce que contient le noeud.

Le type de noeud peut être :

Constante	Valeur	Rôle
ELEMENT_NODE	1	Element
ATTRIBUTE_NODE	2	Attribut

TEXT_NODE	3	
CDATA_SECTION_NODE	4	
ENTITY_REFERENCE_NODE	5	
ENTITY_NODE	6	
PROCESSING_INSTRUCTION_NODE	7	
COMMENT_NODE	8	
DOCUMENT_NODE	9	Racine du document
DOCUMENT_TYPE_NODE	10	
DOCUMENT_FRAGMENT_NODE	11	
NOTATION_NODE	12	

34.1.2. L'interface NodeList

Cette interface définit une liste ordonnée de noeuds suivant l'ordre du document XML. Elle définit deux méthodes :

Méthode	Rôle
int getLength()	Renvoie le nombre de noeuds contenus dans la liste
Node item(int)	Renvoie le noeud dont l'index est fourni en paramètre

34.1.3. L'interface Document

Cette interface définit les caractéristiques pour un objet qui sera la racine de l'arbre DOM. Cette interface hérite de l'interface Node.

Un objet de type Document possède toujours un type de noeud DOCUMENT_NODE.

Méthode	Rôle
DocumentType getDocType()	Renvoyer les informations sur le type de document
Element getDocumentElement()	Renvoyer l'élément racine du document
NodeList getElementsByTagNames(String)	Renvoyer une liste des éléments dont le nom est fourni en paramètre
Attr createAttributes(String)	Créer un attribut dont le nom est fourni en paramètre
CDATASection createCDATASection(String)	Créer un noeud de type CDATA
Comment createComment(String)	Créer un noeud de type commentaire
Element createElement(string)	Créer un noeud de type élément dont le nom est fourni en paramètre

34.1.4. L'interface Element

Cette interface définit des méthodes pour manipuler un élément et en particulier les attributs d'un élément. Un élément dans un document XML correspondant à un tag. L'interface Element hérite de l'interface Node.

Un objet de type Element à toujours pour type de noeud ELEMENT_NODE

Méthode	Rôle
String getAttribute(String)	Renvoyer la valeur de l'attribut dont le nom est fourni en paramètre
removeAttribut(String)	
setAttribut(String, String)	Modifier ou créer un attribut dont le nom est fourni en premier paramètre et la valeur en second
String getTagName()	Renvoyer le nom du tag
Attr getAttributeNode(String)	Renvoyer un objet de type Attr qui encapsule l'attribut dont le nom est fourni en paramètre
Attr removeAttributeNode(Attr)	Supprimer l'attribut fourni en paramètre
Attr setAttributNode(Attr)	Modifier ou créer un attribut
NodeList getElementsByTagName(String)	Renvoyer une liste des noeuds enfants dont le nom correspond au paramètre fourni

34.1.5. L'interface CharacterData

Cette interface définit des méthodes pour manipuler les données de type PCDATA d'un noeud.

Méthode	Rôle
appendData()	Ajouter le texte fourni en paramètre aux données courantes
getData()	Renvoyer les données sous la forme d'une chaîne de caractères
setData()	Permettre d'initialiser les données avec la chaîne de caractères fournie en paramètre

34.1.6. L'interface Attr

Cette interface définit des méthodes pour manipuler les attributs d'un élément.

Les attributs ne sont pas des noeuds dans le modèle DOM. Pour pouvoir les manipuler, il faut utiliser un objet de type Element.

Méthode	Rôle
String getName()	Renvoyer le nom de l'attribut
String getValue()	Renvoyer la valeur de l'attribut
String setValue(String)	Mettre la valeur à celle fournie en paramètre

34.1.7. L'interface Comment

Cette interface permet de caractériser un noeud de type commentaire.

Cette interface étend simplement l'interface CharacterData. Un objet qui implémente cette interface générera un tag de la forme <!-- --> .

34.1.8. L'interface Text

Cette interface permet de caractériser un noeud de type Text. Un tel noeud représente les données d'un tag ou la valeur d'un attribut.

34.2. L'obtention d'un arbre DOM

Pour pouvoir utiliser un arbre DOM représentant un document, il faut utiliser un parseur qui implémente DOM. Ce dernier va parcourir le document XML et créer l'arbre DOM correspondant. Le but est d'obtenir un objet qui implémente l'interface Document car cet objet est le point d'entrée pour toutes opérations sur l'arbre DOM.

Avant la définition de JAXP par Sun, l'instanciation d'un parseur était spécifique à chaque à implémentation.

Exemple : utilisation de Xerces sans JAXP

```
package perso.jmd.tests.testdom;

import org.apache.xerces.parsers.*;
import org.w3c.dom.*;

public class TestDOM2 {

  public static void main(String[] args) {
    Document document = null;
    DOMParser parser = null;

    try {
      parser = new DOMParser();
      parser.parse("test.xml");
      document = parser.getDocument();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

JAXP permet, si le parseur respecte ses spécifications, d'instancier le parseur de façon normalisée.

Exemple : utilisation de Xerces avec JAXP

```
package perso.jmd.tests.testdom;

import org.w3c.dom.*;
import javax.xml.parsers.*;

public class TestDOM1 {

  public static void main(String[] args) {
    Document document = null;
    DocumentBuilderFactory factory = null;

    try {
      factory = DocumentBuilderFactory.newInstance();
      DocumentBuilder builder = factory.newDocumentBuilder();
      document = builder.parse("test.xml");
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

L'utilisation de JAXP est fortement recommandée.

Remarque : JAXP est détaillé dans une des sections suivantes de ce chapitre.

34.3. Le parcours d'un arbre DOM

Cette section sera développée dans une version future de ce document

34.3.1. Les interfaces Traversal

DOM level 2 propose plusieurs interfaces pour faciliter le parcours d'un arbre DOM.

La suite de cette section sera développée dans une version future de ce document

34.4. La modification d'un arbre DOM

Un des grands intérêts du DOM est sa faculté à créer ou modifier l'arbre qui représente un document XML.

34.4.1. La création d'un document

La méthode newDocument() de la classe DocumentBuilder renvoie une nouvelle instance d'un objet de type document qui encapsule un arbre DOM vide.

Il faut a minima ajouter un tag racine au document XML. Pour cela, il faut appeler la méthode createElement() de l'objet Document en lui passant le nom du tag racine pour obtenir une référence sur le nouveau noeud. Il suffit ensuite d'utiliser la méthode appendChild() de l'objet Document en lui fournissant la référence sur le noeud en paramètre.

Exemple :

```
package perso.jmd.tests.testdom;

import org.w3c.dom.*;
import javax.xml.parsers.*;

public class TestDOM09 {

  public static void main(String[] args) {
    Document document = null;
    DocumentBuilderFactory fabrique = null;

    try {
      fabrique = DocumentBuilderFactory.newInstance();
      DocumentBuilder builder = fabrique.newDocumentBuilder();
      document = builder.newDocument();
      Element racine = (Element) document.createElement("bibliotheque");
      document.appendChild(racine);
    } catch (Exception e) {
      e.printStackTrace();
    }
```

```
  }
}
```

34.4.2. L'ajout d'un élément

L'interface Document propose plusieurs méthodes createXXX pour créer des instances de différents types d'éléments. Il suffit alors d'utiliser la méthode appendChild() d'un noeud pour lui attacher un noeud fils.

Exemple :

```
     Element monElement = document.createElement("monelement");
     Element monElementFils = document.createElement("monelementfils");
     monElement.appendChild(monElementFils);
```

Pour ajouter un texte à un noeud, il faut utiliser la méthode createTextNode() pour créer un noeud de type Text et l'ajouter au noeud concerné avec la méthode appendChild().

Exemple :

```
     Element monElementFils = document.createElement("monelementfils");
     monElementFils.appendChild(document.createTextNode("texte du tag fils");
     monElement.appendChild(monElementFils);
```

Pour ajouter un attribut à un élément, il existe deux méthodes : setAttributeNode() et setAtribute().

La méthode setAttributeNode() attend un objet de type Attr qu'il faut préalablement instancier.

Exemple :

```
     Attr monAttribut = document.createAttribute("attribut");
     monAttribut.setValue("valeur");
     monElement.setAttributeNode(monAttribut);
```

La méthode setAttribut permet directement d'associer un attribut en lui fournissant en paramètre son nom et sa valeur.

Exemple :

```
    monElement.setAttribut("attribut","valeur");
```

La création d'un commentaire se fait en utilisant la méthode createComment() de la classe Document.

Toutes ces actions permettent la création complète d'un arbre DOM représentant un document XML.

Exemple : un exemple complet

```
package perso.jmd.tests.testdom;

import org.w3c.dom.*;
import javax.xml.parsers.*;

public class TestDOM11 {

  public static void main(String[] args) {
    Document document = null;
    DocumentBuilderFactory fabrique = null;

    try {
```

```
      fabrique = DocumentBuilderFactory.newInstance();
      DocumentBuilder builder = fabrique.newDocumentBuilder();
      document = builder.newDocument();
      Element racine = (Element) document.createElement("bibliotheque");
      document.appendChild(racine);
      Element livre = (Element) document.createElement("livre");
      livre.setAttribute("style", "1");
      Attr attribut = document.createAttribute("type");
      attribut.setValue("broche");
      livre.setAttributeNode(attribut);
      racine.appendChild(livre);
      livre.setAttribute("style", "1");
      Element titre = (Element) document.createElement("titre");
      titre.appendChild(document.createTextNode("Titre 1"));
      livre.appendChild(titre);
      racine.appendChild(document.createComment("mon commentaire"));
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<bibliotheque>
    <livre style="1" type="broche">
        <titre>Titre 1</titre>
    </livre>
    <!--mon commentaire-->
</bibliotheque>
```

34.5. L'envoie d'un arbre DOM dans un flux

Une fois un arbre DOM créé ou modifié, il est souvent utile de l'envoyer dans un flux (sauvegarde dans un fichier ou une base de données, envoi dans un message JMS ...).

Bizarrement, DOM level 1 et 2 ne propose rien pour réaliser cette tache quasiment obligatoire à effectuer. Ainsi, chaque implémentation propose sa propre méthode en attendant des spécifications qui feront sûrement partie du DOM Level 3.

34.5.1. Un exemple avec Xerces

Xerces fournit la classe XMLSerializer qui permet de créer un document XML à partir d'un arbre DOM.

Xerces est téléchargeable sur le site web http://xml.apache.org/xerces2-j/ sous la forme d'une archive de type zip qu'il faut décompresser dans un répertoire du système. Il suffit alors d'ajouter les fichiers xmlParserAPIs.jar et xercesImpl.jar dans le classpath.

Exemple :

```
package perso.jmd.tests.testdom;
import org.w3c.dom.*;
import javax.xml.parsers.*;
import org.apache.xml.serialize.*;

public class TestDOM10 {

  public static void main(String[] args) {
    Document document = null;
    DocumentBuilderFactory fabrique = null;

    try {
```

```
      fabrique = DocumentBuilderFactory.newInstance();
      DocumentBuilder builder = fabrique.newDocumentBuilder();
      document = builder.newDocument();
      Element racine = (Element) document.createElement("bibliotheque");
      document.appendChild(racine);

      for (int i = 1; i < 4; i++) {
        Element livre = (Element) document.createElement("livre");
        Element titre = (Element) document.createElement("titre");
        titre.appendChild(document.createTextNode("Titre "+i));
        livre.appendChild(titre);
        Element auteur = (Element) document.createElement("auteur");
        auteur.appendChild(document.createTextNode("Auteur "+i));
        livre.appendChild(auteur);
        Element editeur = (Element) document.createElement("editeur");
        editeur.appendChild(document.createTextNode("Editeur "+i));
        livre.appendChild(editeur);
        racine.appendChild(livre);
      }

      XMLSerializer ser = new XMLSerializer(System.out,
         new OutputFormat("xml", "UTF-8", true));
      ser.serialize(document);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<bibliotheque>
    <livre>
        <titre>Titre 1</titre>
        <auteur>Auteur 1</auteur>
        <editeur>Editeur 1</editeur>
    </livre>
    <livre>
        <titre>Titre 2</titre>
        <auteur>Auteur 2</auteur>
        <editeur>Editeur 2</editeur>
    </livre>
    <livre>
        <titre>Titre 3</titre>
        <auteur>Auteur 3</auteur>
        <editeur>Editeur 3</editeur>
    </livre>
</bibliotheque>
```

35. XSLT (Extensible Stylesheet Language Transformations)

XSLT est une recommandation du consortium W3C qui permet de transformer facilement des documents XML en d'autres documents standard sans programmation. Le principe est de définir une feuille de style qui indique comment transformer le document XML et de le fournir avec le document à un processeur XSLT.

On peut produire des documents de différents formats : XML, HTML, XHTML, WML, PDF, etc...

XSLT fait parti de XSL avec les recommandations :

- XSL-FO : flow object
- XPath : langage pour spécifier un élément dans un document. Ce langage est utilisé par XSL.

Une feuille de style XSLT est un fichier au format XML qui contient les informations nécessaires au processeur pour effectuer la transformation.

Le composant principal d'une feuille de style XSLT est le template qui définit le moyen de transformer un élément du document XML dans le nouveau document.

XSLT est très relativement complet et complexe : cette section n'est qu'une présentation rapide de quelques fonctionnalités de XSLT. XSLT possède plusieurs fonctionnalités avancées, telles que la sélection des éléments à traiter, filtrer des éléments ou trier les éléments.

35.1. XPath

XML Path ou XPath est une spécification qui fournit une syntaxe pour permettre de sélectionner un ou plusieurs éléments dans un document XML. Il existe sept types d'éléments différents :

- racine (root)
- element
- text
- attribute (attribute)
- commentaire (comment)
- instruction de traitement (processing instruction)
- espace de nommage (name space)

Cette section ne présente que les fonctionnalités de base de XPath.

XPath est utilisé dans plusieurs technologies liées à XML tel que XPointer et XSLT.

Un document XML peut être représenté sous la forme d'un arbre composé de noeuds. XPath grâce à une notation particulière permet de localiser précisément un composant de l'arbre.

La notation reprend une partie de la notation utilisée pour naviguer dans un système d'exploitation, ainsi :

* le séparateur est le caractère slash /
* pour préciser un chemin à partir de la racine (chemin absolu), il faut qu'il commence par un /
* un double point .. permet de préciser l'élément père de l'élément courant
* un simple point . permet de préciser l'élément courant
* un arobase @ permet de préciser un attribut d'un élément
* pour préciser l'indice d'un élément il faut le préciser entre crochets

XPath permet de filtrer les éléments sur différents critères en plus de leur nom

* @categorie="test" : recherche un attribut dont le nom est categorie est dont la valeur est "test"
* une barre verticale | permet de préciser deux valeurs

35.2. La syntaxe de XSLT

Une feuille de style XSLT est un document au format XML. Il doit donc respecter toutes les règles d'un tel document. Pour préciser les différentes instructions permettant de réaliser la transformation, un espace de nommage particulier est utilisé : xsl. Tous les tags de XSLT commencent donc par ce préfixe, ainsi le tag racine du document est xsl:stylesheet. Ce tag racine doit obligatoirement posséder un attribut version qui précise la version de XSLT utilisée.

Exemple : une feuille de style minimale qui ne fait rien

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

</xsl:stylesheet>
```

Le tag xsl:output permet de préciser le format de sortie. Ce tag possède plusieurs attributs :

* method : cet attribut permet de préciser le format. Les valeurs possibles sont : texte, xml ou html
* indent : cet attribut permet de définir si la sortie doit être indentée ou non. Les valeurs possibles sont : yes ou no
* encoding : cet attribut permet de préciser le jeu de caractères utilisé pour la sortie

Pour effectuer la transformation, le document doit contenir des règles. Ces règles suivent une syntaxe particulière et sont contenues dans des modèles (templates) associés à un ou plusieurs éléments désignés avec un motif au format XPath.

Un modèle est défini grâce au tag xsl:template. La valeur de l'attribut match permet de fournir le motif au format XPath qui sélectionnera le ou les éléments sur lequel le modèle va agir.

Le tag xsl:apply-templates permet de demander le traitements des autres modèles définis pour chacun des noeuds fils du noeud courant.

Le tag xsl:value-of permet d'extraire la valeur de l'élément respectant le motif XPath fourni avec l'attribut select.

Il existe beaucoup d'autres tags notamment plusieurs qui permettent d'utiliser des structures de contrôles de type itératifs ou conditionnels.

Le tag xsl:for-each permet de parcourir un ensemble d'élément sélectionnés par l'attribut select. Le modèle sera appliqué sur chacun des éléments de la liste

Le tag xsl:if permet d'exécuter le modèle si la condition précisée par l'attribut test au format XPath est juste. XSLT ne définit pas de tag équivalent à la partie else : il faut définir un autre tag xsl:if avec une condition opposée.

Le tag xsl:choose permet de définir plusieurs conditions. Chaque condition est précisée grâce à l'attribut xsl:when avec l'attribut test. Le tag xsl:otherwise permet de définir un cas par défaut qui ne correspond aux autres cas définis dans le tag xsl:choose.

Le tag xsl:sort permet de trier un ensemble d'éléments. L'attribut select permet de préciser les éléments qui doivent être triés. L'attribut data-type permet de préciser le format des données (text ou number). l'attribut order permet de préciser l'ordre de tri (ascending ou descending).

35.3. Un exemple avec Internet Explorer

Le plus simple pour tester une feuille XSLT qui génère une page HTML est de la tester avec Internet Explorer version 6. Cette version est entièrement compatible avec XML et XSLT. Les versions 5 et 5.5 ne sont que partiellement compatibles. Les versions antérieures ne le sont pas du tout.

35.4. Un exemple avec Xalan 2

Xalan 2 utilise l'API JAXP.

Exemple : TestXSL2.java

```
import javax.xml.transform.*;
import javax.xml.transform.stream.*;
import org.xml.sax.*;
import java.io.IOException;

public class TestXSL2
{
  public static void main(String[] args)
        throws TransformerException, TransformerConfigurationException,
         SAXException, IOException
        {

    TransformerFactory tFactory = TransformerFactory.newInstance();
    Transformer transformer = tFactory.newTransformer(new StreamSource("test.xsl"));

    transformer.transform(new StreamSource("test.xml"), new StreamResult("test.htm"));
  }
}
```

Exemple : la feuille de style XSL test.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns="http://www.w3.org/TR/REC-html40">

<xsl:output method="html" indent="no" />
<xsl:strip-space elements="*" />

<xsl:template match="/">
<HTML>
<HEAD>
<TITLE>Test avec XSL</TITLE>
</HEAD>
<xsl:apply-templates />
</HTML>
</xsl:template>

<xsl:template match="BIBLIOTHEQUE">
<BODY>
<H1>Liste des livres</H1>
<TABLE border="1" cellpadding="4">
<TR><TD>Titre</TD><TD>Auteur</TD><TD>Editeur</TD></TR>
<xsl:apply-templates />
</TABLE>
</BODY>
</xsl:template>

<xsl:template match="LIVRE">
<TR>
<TD><xsl:apply-templates select="TITRE"/></TD>
<TD><xsl:apply-templates select="AUTEUR"/></TD>
<TD><xsl:apply-templates select="EDITEUR"/></TD>
</TR>
```

```
</xsl:template>
</xsl:stylesheet>
```

Exemple : compilation et execution avec Xalan

```
javac TestXSL2.java -classpath .;xalan2.jar
java -cp .;xalan2.jar TestXSL2
```

La suite de cette section sera développée dans une version future de ce document

36. Les modèles de document

36.1. L'API JDOM

JDOM est une API open source Java dont le but est de représenter et manipuler un document XML de manière intuitive pour un développeur Java sans requérir une connaissance pointue de XML. Par exemple, JDOM utilise des classes plutôt que des interfaces. Ainsi pour créer un nouvel élément, il faut simplement instancier une classe.

Malgré la similitude de nom entre JDOM et DOM, ces deux API sont très différentes. JDOM est une API uniquement Java car elle s'appuie sur un ensemble de classes de l'API Java notamment celles de l'API Collection.

Le site officiel de l'API est à l'url http://www.jdom.org/

36.1.1. L'historique de JDOM

En 2000, Brett McLaughlin et Jason Hunter développent une nouvelle API dédiée aux traitements de documents XML en Java. Le but est de fournir une API plus conviviale à utiliser en Java que SAX ou DOM.

L'historique de JDOM est marquée par plusieurs versions béta et stables :

- La version betâ 3 est diffusée en avril 2000.
- La version 1.0 a été publié en septembre 2004.
- La version 1.1 a été publiée en novembre 2007.

JDOM a fait l'objet d'une spécification sous la Java Specification Request numéro 102 (JSR-102) : malheureusement celle-ci n'a pas aboutie.

36.1.2. La présentation de JDOM

Le but de JDOM n'est pas de définir un nouveau type de parseur mais de faciliter la manipulation au sens large de document XML : lecture d'un document, représentation sous forme d'arborescence, manipulation de cet arbre, définition d'un nouveau document, exportation vers plusieurs formats cibles ...

Dans le rôle de manipulation sous forme d'arbre, JDOM possède moins de fonctionnalités que DOM mais en contre partie il offre une plus grande facilité pour répondre aux cas les plus classiques d'utilisation.

Cette facilité d'utilisation de JDOM lui permet d'être une API dont l'utilisation est assez répandue.

JDOM est donc un modèle de documents objets open source dédié à Java pour encapsuler un document XML. JDOM propose aussi une intégration de SAX, DOM, XSLT et XPath.

JDOM n'est pas un parseur : il a d'ailleurs besoin d'un parseur externe de type SAX ou DOM pour analyser un document et créer la hiérarchie d'objets relative à un document XML. L'utilisation d'un parseur de type SAX est recommandée car

elle consomme moins de ressources que DOM pour cette opération. Par défaut, JDOM utilise le parseur défini via JAXP.

Un document XML est encapsulé dans un objet de type Document qui peut contenir des objets de type Comment, ProcessingInstruction et l'élément racine du document encapsulé dans un objet de type Element.

Les éléments d'un document sont encapsulés dans des classes dédiées : Element, Attribute, Text, ProcessingInstruction, Namespace, Comment, DocType, EntityRef, CDATA.

Un objet de type Element peut contenir des objets de type Comment, Text et d'autres objets de type Element.

A l'exception des objets de type Namespace, les éléments sont créés en utilisant leur constructeur.

JDOM vérifie que les données contenues dans les éléments respectent la norme XML : par exemple, il n'est pas possible de créer un commentaire contenant deux caractères moins qui se suivent.

Une fois un document XML encapsulé dans un arbre d'objets, il est possible de modifier cet arbre dans le respect des spécifications de XML.

JDOM permet d'exporter un arbre d'objets d'un document XML dans un flux, un arbre DOM ou un ensemble d'événements SAX.

JDOM interagit donc avec SAX et DOM pour créer un document en utilisant ces parseurs ou exporter un document vers ces API, ce qui permet de facilement intégrer JDOM dans des traitements existants. JDOM propose cependant sa propre API.

36.1.3. Les fonctionnalités et les caractéristiques

JDOM propose plusieurs fonctionnalités :

- Création de documents XML
- Encapsulation d'un document XML sous la forme d'objets Java de l'API
- Exportation d'un document dans un fichier, un flux SAX ou un arbre DOM
- Support de XSLT
- Support de XPath

Les points caractéristiques de l'API JDOM sont :

- elle est développée spécifiquement en et pour Java en utilisant les fonctionnalités de Java au niveau syntaxique et sémantique (utilisation des collections de Java 2, de l'opérateur new pour instancier des éléments, redéfinition des méthodes equals(), hashCode(), toString(), implémentation des interfaces Cloneable et Serializable, ...)
- elle se veut intuitive et productive notamment grâce des classes dédiées à chaque élément instancié via leur constructeur et l'utilisation de getter/setter
  Exemple pour obtenir le texte d'un élément
  DOM : String content = element.getFirstChild().getValue();
  JDOM : String text = element.getText();
- elle se veut rapide et légère
- elle veut masquer la complexité de certains aspects de XML tout en respectant ses spécifications
- elle doit permettre les interactions entre SAX et DOM. JDOM peut encapsuler un document XML dans un hiérarchie d'objets à partir d'un flux, d'un arbre DOM ou d'événements SAX. Il est aussi capable d'exporter un document dans ces différents formats.

Il est légitime de se demander qu'elle est l'utilité de proposer une nouvelle API pour manipuler des documents XML en Java alors que plusieurs standards existent déjà. En fait le besoin est réel car JDOM propose des réponses à certaines faiblesses de SAX et DOM.

DOM est une API indépendante de tout langage : son implémentation en Java ne tient donc pas compte des spécificités et standards de Java ce qui rend sa mise en oeuvre peu aisée. JDOM est plus intuitif et facile à mettre en oeuvre que DOM.

Comme DOM, JDOM encapsule un document XML entier dans un arbre d'objets. Par contre chaque élément du document est encapsulé dans une classe dédiée selon son type et non sous la forme d'un objet de type Node.

JDOM peut être utilisé comme une alternative à DOM pour manipuler un document XML. JDOM n'est pas un remplaçant à DOM puisque ce n'est pas un parseur et il propose des interactions avec DOM en entrée et en sortie.

L'utilisation de DOM requiert de nombreuses ressources notamment à cause de son API qui de surcroit n'est pas intuitive en Java puisque développé de façon indépendant de tout langage et que son organisation est proche de celle des spécifications XML (tous les éléments sont des Nodes par exemple).

SAX est particulièrement bien adapté à la lecture rapide avec peu de ressources d'un document XML mais son modèle de traitement par événements n'est pas intuitive et surtout SAX ne permet de modifier ni de naviguer dans un document.

JDOM propose d'apporter une solution à ces différents problèmes dans une seule et même API.

Afin de rendre les exemples plus clairs, la plupart des exemples de ce chapitre hérite de la classe ci-dessous qui propose une méthode pour exporter un document sur la console.

Exemple :

```
package com.jmdoudoux.test.jdom;

import org.jdom.Document;
import org.jdom.output.Format;
import org.jdom.output.XMLOutputter;

public abstract class TestJDOM {

  protected static void afficher(Document document)
  {
     try
     {
        XMLOutputter sortie = new XMLOutputter(Format.getPrettyFormat());
        sortie.output(document, System.out);
     }
     catch (java.io.IOException e){}
  }

}
```

36.1.4. L'installation de JDOM

Avant la version 1.0 de JDOM, il est nécessaire de compiler l'API avec un script Ant fourni.

A partir de la version 1.0, l'archive de JDOM contient directement un binaire de la bibliothèque utilisable.

36.1.4.1. L'installation de JDOM Betâ 7 sous Windows

Pour utiliser JDOM il faut construire la bibliothèque grâce à l'outil Ant. Ant doit donc être installé sur la machine.

Il faut aussi que la variable JAVA_HOME soit positionnée avec le répertoire qui contient le JDK.

Exemple :

```
set JAVA_HOME=c:\j2sdk1.4.0-rc
```

Il suffit alors simplement d'exécuter le fichier build.bat situé dans le répertoire d'installation de JDom.

Un message informe de la fin de la compilation :

Exemple :

```
package:

[jar] Building jar: C:\java\jdom-b7\build\jdom.jar
BUILD SUCCESSFUL
Total time: 1 minutes 33 seconds
```

Le fichier jdom.jar est créé dans le répertoire build.

Pour utiliser JDOM dans un projet, il faut obligatoirement avoir un parseur XML SAX et/ou DOM. Pour les exemples de cette section, lorsqu'aucun parser n'est fourni avec le JDK, c'est Xerces qui est utilisé. Il faut aussi avoir le fichier jaxp.jar.

Pour compiler et exécuter les exemples de cette section, j'ai utilisé le script suivant :

Exemple :

```
javac %1.java -classpath .;jdom.jar;xerces.jar;jaxp.jar
java -classpath .;jdom.jar;xerces.jar;jaxp.jar %1
```

36.1.4.2. L'installation de la version 1.x

L'archive contenant JDOM peut être téléchargée à l'url http://www.jdom.org/dist/binary/

Il suffit de décompresser le contenu de l'archive dans un répertoire du système et d'ajouter le fichier jdom.jar contenu dans le sous répertoire build au classpath.

36.1.5. Les différentes entités de JDOM

Pour traiter un document XML, JDOM définit plusieurs entités qui peuvent être regroupées en trois groupes :

- les éléments de l'arbre
    - le document : la classe Document
    - les éléments : la classe Element
    - les commentaires : la classe Comment
    - les attributs : la classe Attribute
    - etc ...
- les entités pour obtenir un parseur :
    - les classes SAXBuilder et DOMBuilder
- les entités pour produire un document
    - les classes XMLOutputter, SAXOutputter, DOMOutputter

Ces classes sont regroupées dans cinq packages :

- org.jdom
- org.jdom.adapters
- org.jdom.input
- org.jdom.output
- org.jdom.transform

Attention : cette API a énormément évoluée jusqu'à sa version 1.0. Beaucoup de méthodes ont été déclarées deprecated au fur et à mesure des différentes versions béta.

36.1.5.1. La classe Document

La classe org.jdom.Document encapsule l'arbre dans lequel JDOM stocke le document XML. Pour obtenir un objet Document, il y a deux possibilités :

- utiliser un objet XXXBuilder qui va parser un document XML existant et créer l'objet Document en utilisant un parseur
- instancier un nouvel objet Document pour créer un nouveau document XML

Pour créer un nouveau document, il suffit d'instancier un objet Document en utilisant un des constructeurs fournis dont les principaux sont :

Constructeur	Rôle
Document()	
Document(Element)	Création d'un document avec l'élément racine fourni
Document(Element, DocType)	Création d'un document avec l'élément racine et la déclaration doctype fournie
Document(List)	Création d'un document avec les entités fournies (élément racine, commentaires, instructions de traitement)
Document(List, DocType)	Création d'un document avec les entités et le type de document fournis

Exemple :

```
import org.jdom.*;

public class TestJDOM2 {
        public static void main(String[] args) {
               Element racine = new Element("bibliothèque");
               Document document = new Document(racine);

        }
}
```

La classe Document possède plusieurs méthodes dont les principales sont :

Méthode	Rôle
Document addContent(Comment)	Ajouter un commentaire au document
List getContent()	Renvoyer un objet List qui contient chaque élément du document
DocType getDocType()	Renvoyer un objet contenant les caractéristiques Doctype du document
Document setDocType()	Définir le DocType du document
Element getRootElement()	Renvoyer l'élément racine du document
Document setRootElement(Element)	Définir l'élément racine du document
boolean hasRootElement()	Renvoyer un booléen qui indique si le document possède un élément racine
Element detachRootElement()	Détache l'élément racine du document
Document addContent(ProcessingInstruction)	Ajouter une instruction de traitement
Document addContent(Comment)	Ajouter un commentaire
Document removeContent(ProcessingInstruction)	Supprimer une instruction de traitement
Document removeContent(Comment)	Supprimer un commentaire

Un objet de type Document possède :

- Un élément racine (RootElement)
- Un objet de type DocType facultatif
- Une collection des éléments rattachés au document : l'élément racine, et éventuellement des instructions de traitement et des commentaires.

Un document peut ne pas avoir d'élément racine, lorsqu'il est créé avec le constructeur par défaut mais dans ce cas le document n'est utilisable qu'à partir du moment où l'élément racine est ajouté au document.

Pour obtenir un document à partir d'un document XML existant, JDOM propose les classes SAXBuilder et DOMbuilder du package org.jdom.input.

36.1.5.2. La classe DocType

JDOM n'offre pas un modèle complet d'objets pour le support des DTD : seule la classe DocType est proposée pour encapsuler la déclaration d'un type document.

Pour instancier un objet de type DocType, il suffit d'utiliser un de ces constructeurs.

L'association de la DTD au document peut se faire en utilisant le constructeur de la classe Document qui attend un tel objet en paramètre ou en utilisant la méthode setDocType de la classe Document.

Exemple :

```
package com.jmdoudoux.test.jdom;

import org.jdom.DocType;
import org.jdom.Document;
import org.jdom.Element;

public class TestJDOM4 extends TestJDOM {

  public static void main(
      String[] args) {

    Element racine = new Element("bibliotheque");
    DocType docType = new DocType("bibliotheque", "bibliotheque.dtd");
    Document document = new Document(racine, docType);

    afficher(document);
  }
}
```

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE bibliotheque SYSTEM "bibliotheque.dtd">

<bibliotheque />
```

La classe DocType n'encapsule que des données informatives sur la DTD dans 4 propriétés :

- root element name
- internal DTD subset
- system ID
- public ID

Exemple de déclaration :

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

Dans cet exemple :

- root element name : html
- system ID : http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd
- public ID : -//W3C//DTD XHTML 1.0 Transitional//EN

Exemple :

```
package com.jmdoudoux.test.jdom;

import org.jdom.DocType;
import org.jdom.Document;
import org.jdom.Element;

public class TestJDOM6 extends TestJDOM {

  public static void main(
      String[] args) {

    Element racine = new Element("html");
    DocType docType = new DocType("html", "-//W3C//DTD XHTML 1.0 Transitional//EN",
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd");

    Document document = new Document(racine, docType);

    afficher(document);
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html />
```

JDOM permet de déclarer une DTD mais ne l'utilise en aucune façon pour assurer la validité du document. JDOM vérifie simplement que le document est bien formé.

Exemple :

```
package com.jmdoudoux.test.jdom;

import org.jdom.DocType;
import org.jdom.Document;
import org.jdom.Element;

public class TestJDOM7 extends TestJDOM {

  public static void main(
      String[] args) {

    Element racine = new Element("test");
    DocType docType = new DocType("html", "-//W3C//DTD XHTML 1.0 Transitional//EN",
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd");

    Document document = new Document(racine, docType);

    afficher(document);
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<test />
```

Pour définir une DTD directement dans la déclaration, il faut utiliser la méthode setInternalSubset() en lui passant en paramètre la chaîne de caractères contenant la DTD.

Pour associer un document de type DocType à un document, il faut utiliser un des constructeurs de la classe Document attendant un tel objet en paramètre ou utiliser la méthode setDocType() de la classe Document.

Remarque : JDOM ne permet pas de valider un document en mémoire.

Pour vérifier la validité d'un document, il faut exporter le document et utiliser un parseur en activant l'option de validation.

36.1.5.3. La classe Element

La structure d'un document XML est composée d'éléments encapsulés dans la classe org.jdom.Element.

La classe Element encapsule un élément du document. Un élément peut contenir du texte, des attributs, des commentaires, et tous les autres éléments définis par la norme XML.

Cette classe possède plusieurs constructeurs dont les principaux sont :

Constructeur	Rôle
Element(String)	Créer un élément, en précisant son nom
Element(String, Namespace)	Créer un élément, en précisant son nom et son espace de nommage
Element(String, String)	Créer un objet, en précisant son nom et l'URI de son espace de nommage
Element(String, String, String)	Créer un objet, en précisant son nom et le préfixe et l'URI de son espace de nommage

La classe Element possède plusieurs propriétés :

- Name : le nom de l'élément
- Namespace : l'espace de nommage de l'élément. Il y a toujours un objet de type Namespace pour chaque élément : si l'élément ne possède pas d'espace de nommage alors la propriété vaut Namespace.NO_NAMESPACE.
- Content : collection de type List des éléments fils de l'élément
- Parent : élément père de l'élément : peut être null si l'élément est l'élément racine ou si l'élément n'est pas ajouté dans un document. Cette propriété ne peut être modifiée directement. Elle est modifiée par la méthode addContent() lors de l'association de l'élément à son père. Cette association n'est possible que pour un élément qui n'a pas déjà un père.
- Document : document qui contient cet élément : peut être null si l'élément n'est pas ajouté à un document.
- Attributes : une collection de type List qui encapsule les attributs de l'élément

La classe Element possède de nombreuses méthodes pour obtenir, ajouter ou supprimer une entité de l'élément (un élément enfant, le texte, un attribut, un commentaire, ...) :

Méthode	Rôle

Element addContent(Comment)	Ajouter un commentaire à l'élément
Element addContent(Element)	Ajouter un élément fils à l'élément
Element addContent(String text)	Ajouter des données sous forme de texte à l'élément
Attribute getAttribute(String)	Renvoyer l'attribut dont le nom est fourni en paramètre
Attribute getAttribute(String, Namespace)	Renvoyer l'attribut dont le nom et l'espace de nommage sont fournis en paramètres
List getAttributes()	Renvoyer une collection qui contient tous les attributs
String getAttributeValue(String)	Renvoyer la valeur de l'attribut dont le nom est fourni en paramètres
String getAttributeValue(String, Namespace)	Renvoyer la valeur de l'attribut dont le nom et l'espace de nommage sont fournis en paramètres
Element getChild(String)	Renvoyer le premier élément enfant dont le nom est fourni en paramètres
Element getChild(String, Namespace)	Renvoyer le premier élément enfant dont le nom et l'espace de nommage sont fournis en paramètres
List getChildren()	Renvoyer une liste qui contient tous les éléments enfants directement rattachés à l'élément
List getChildren(String)	Renvoyer une liste qui contient tous les éléments enfants directement rattachés à l'élément dont le nom est fourni en paramètre
List getChildren(String, Namespace)	Renvoyer une liste qui contient tous les éléments enfants directement rattachés à l'élément dont le nom et l'espace de nommage sont fournis en paramètre
String getChildText(String name)	Renvoyer le texte du premier élément enfant dont le nom est fourni en paramètre
String getChildText(String, Namespace)	Renvoyer le texte du premier élément enfant dont le nom et l'espace de nommage sont fournis en paramètre
List getContent()	Renvoyer une liste qui contient toutes les entités de l'élément (texte, élément, commentaire ...)
Document getDocument()	Renvoyer l'objet Document qui contient l'élément
Element getParent()	Renvoyer l'élément père de l'élément
String getText()	Renvoyer les données au format texte contenues dans l'élément
boolean hasChildren()	Renvoyer un élément qui indique si l'élément possède des éléments fils
boolean isRootElement()	Renvoyer un booléen qui indique si l'élément est l'élément racine du document
boolean removeAttribute(String)	Supprimer l'attribut dont le nom est fourni en paramètre
boolean removeAttribute(String, Namespace)	Supprimer l'attribut dont le nom et l'espace de nommage sont fournis en paramètres
boolean removeChild(String)	Supprimer l'élément enfant dont le nom est fourni en paramètre
boolean removeChild(String, Namespace)	Supprimer l'élément enfant dont le nom et l'espace de nommage sont fournis en paramètres
boolean removeChildren()	Supprimer tous les éléments enfants
boolean removeChildren(String)	Supprimer tous les éléments enfants dont le nom est fourni en paramètre
boolean removeChildren(String, Namespace)	Supprimer tous les éléments enfants dont le nom et l'espace de nommage sont fournis en paramètres
boolean removeContent(Comment)	Supprimer le commentaire fourni en paramètre
boolean removeContent(Element)	Supprimer l'élément fourni en paramètre
Element setAttribute(Attribute)	Ajouter un attribut

Element setAttribute(String, String)	Ajouter un attribut dont le nom et la valeur sont fournis en paramètres
Element setAttribute(String, String, Namespace)	Ajouter un attribut dont le nom, la valeur et l'espace de nommage sont fournis en paramètres
Element setText(String)	Mettre à jour les données au format texte de l'élément

Pour obtenir l'élément racine d'un document, il faut utiliser la méthode getRootElement() de la classe Document. Celle ci renvoie un objet de type Element. Il est ainsi possible d'utiliser les méthodes ci dessous pour parcourir et modifier le contenu du document.

L'utilisation de la classe Elément est très facile.

Pour créer un élément, il suffit d'instancier un objet de type Element.

Exemple :

```
          Element element = new Element("element1");
          element.setAttribute("attribut1","valeur1");
          element.setAttribute("attribut2","valeur2");
```

La classe possède plusieurs méthodes pour obtenir les entités de l'élément, un élément fils particulier ou une liste d'élément fils. Un appel successif à ces méthodes permet d'obtenir un élément précis du document.

Les méthodes de type setter, qui devraient par convention ne rien retourner (void), renvoie l'instance de type Element lui-même. Ceci permet de chaîner les appels aux différents setters.

Exemple :

```
package com.jmdoudoux.test.jdom;

import org.jdom.Document;
import org.jdom.Element;

public class TestJDOM28 extends TestJDOM {

  public static void main(
      String[] args) {

    Element racine = new Element("bibliotheque");
    Document document = new Document(racine);
    Element livres = new Element("livres");
    racine.addContent(livres);
    livres.addContent(new Element("livre")
    .addContent(new Element("titre").setText("Titre livre 1"))
    .addContent(new Element("auteur").setText("Auteur 1"))
    );

    afficher(document);
  }
}
```

Résultat :

```
</a><?xml version="1.0" encoding="UTF-8"?>
<bibliotheque>
  <livres>
    <livre>
      <titre>Titre livre 1</titre>
      <auteur>Auteur 1</auteur>
    </livre>
  </livres>
</bibliotheque>
```

36.1.5.4. La classe Attribut

La classe org.jdom.Attribut encapsule un attribut d'un élément.

La classe Attribut possède plusieurs propriétés :

- name : le nom de l'attribut
- namespace : l'espace de nommage
- value : la valeur de l'attribut
- parent : l'élément qui contient l'attribut
- Type : le type de l'attribut (par défaut Attribute.UNDECLARED_ATTRIBUTE)

La classe Element propose les méthodes getAttribute() et getAttributeValue() pour obtenir un attribut ou la valeur d'un attribut sous la forme d'un objet de type String. La méthode getAttribute() renvoie null si l'attribut n'existe pas.

La classe Attribute propose plusieurs méthodes getXXXValue() qui tentent de fournir la valeur de l'attribut dans le type primitif XXX. Si la conversion échoue, alors une exception de type org.jdom.DataConversionException est levée.

Exemple :

```
package com.jmdoudoux.test.jdom;

import java.io.File;
import java.io.IOException;

import org.jdom.Attribute;
import org.jdom.DataConversionException;
import org.jdom.Document;
import org.jdom.Element;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;

public class TestJDOM21 extends TestJDOM {

  public static void main(
      String[] args) {

    try {
      SAXBuilder builder = new SAXBuilder();
      Document document = builder.build(new File("test.xml"));

      afficher(document);

      Element table = document.getRootElement().getChild("body").getChild("table");
      System.out.println("attribut width : " + table.getAttributeValue("width"));

      System.out.println("");
      Attribute border = table.getAttribute("border");
      System.out.println("attribut " + border.getName() + " : " + border.getValue());
      System.out.println("attribut " + border.getName() + " : " + border.getIntValue());

      System.out.println("");
      Attribute width = table.getAttribute("width");
      try {
        System.out.println("attribut " + width.getName()
          + " : " + width.getIntValue());
      } catch (DataConversionException dce) {
        System.out.println("attribut " + width.getName()
          + " : impossible d'obtenir une valeur entière");
      }

      System.out.println("");
      Attribute cellspacing = table.getAttribute("cellspacing");
      if (cellspacing == null) {
        System.out.println("l'attribut cellspacing n'existe pas");
      }

    } catch (JDOMException e) {
```

```
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<html>
  <head />
  <body>
    <table width="100%" border="0" />
  </body>
</html>

attribut width : 100%

attribut border : 0
attribut border : 0

attribut width : impossible d'obtenir une valeur entière

l'attribut cellspacing n'existe pas
```

L'association d'un attribut à un élément se fait généralement en utilisant les méthodes setAttribute() et removeAttribute() de la classe Element plutôt qu'en manipulant des instances de la classe Attribute.

Exemple :

```
package com.jmdoudoux.test.jdom;

import org.jdom.Attribute;
import org.jdom.Document;
import org.jdom.Element;

public class TestJDOM22 extends TestJDOM {

  public static void main(
      String[] args) {

    Element racine = new Element("html");
    Document document = new Document(racine);
    Element body = new Element("body");
    racine.addContent(body);
    Element table = new Element("table");
    body.addContent(table);
    table.setAttribute("width","100%");
    table.setAttribute(new Attribute("border","0"));
    table.setAttribute("cellspacing","10");
    table.removeAttribute("cellspacing");

    afficher(document);
  }
}
```

Résultat :

```
 <?xml version="1.0" encoding="UTF-8"?>
<html>
  <body>
    <table width="100%" border="0" />
  </body>
</html>
```

JDom vérifie les valeurs fournies pour la propriété Name et Value.

Exemple :

```
package com.jmdoudoux.test.jdom;

import org.jdom.Document;
import org.jdom.Element;

public class TestJDOM24 extends TestJDOM {

  public static void main(
      String[] args) {

    Element racine = new Element("html");
    Document document = new Document(racine);
    Element body = new Element("body");
    racine.addContent(body);
    Element table = new Element("table");
    body.addContent(table);
    table.setAttribute("@valeur","100");

    afficher(document);
  }
}
```

Résultat :

```
Exception in thread "main" org.jdom.IllegalNameException:
The name "@valeur" is not legal for JDOM/XML attributes:
XML names cannot begin with the character "@".
        at org.jdom.Attribute.setName(Attribute.java:363)
        at org.jdom.Attribute.<init>(Attribute.java:227)
        at org.jdom.Attribute.<init>(Attribute.java:251)
        at org.jdom.Element.setAttribute(Element.java:1128)
        at com.jmdoudoux.test.jdom.TestJDOM24.main(TestJDOM24.java:21)
```

La classe Element propose la méthode getAttributes() qui renvoie une collection d'objets de type Attribute contenant tous les attributs de l'élément.

Pour supprimer tous les attributs d'un élément, il suffit d'utiliser la méthode clear() sur la collection.

Exemple :

```
package com.jmdoudoux.test.jdom;

import java.util.List;
import java.util.ListIterator;

import org.jdom.Attribute;
import org.jdom.Document;
import org.jdom.Element;

public class TestJDOM23 extends TestJDOM {

  public static void main(
      String[] args) {

    Element racine = new Element("html");
    Document document = new Document(racine);
    Element body = new Element("body");
    racine.addContent(body);
    Element table = new Element("table");
    body.addContent(table);
    table.setAttribute("width","100%");
    table.setAttribute(new Attribute("border","0"));
    table.setAttribute("cellspacing","10");
```

```
    List attributs = table.getAttributes();
    ListIterator iterator = attributs.listIterator();

    System.out.println("Liste des attributs");
    while (iterator.hasNext()) {
      Attribute attribut = (Attribute) iterator.next();
      System.out.println("attribut "+attribut.getName()+" : "+attribut.getValue());
    }

    System.out.println();
    // supprimer tous les attributs
    attributs.clear();

    afficher(document);
  }
}
```

Résultat :

```
Liste des attributs
attribut width : 100%
attribut border : 0
attribut cellspacing : 10

<?xml version="1.0" encoding="UTF-8"?>
<html>
  <body>
    <table />
  </body>
</html>
```

Un objet de type Attribut ne peut avoir qu'un seul élément parent.

Exemple :

```
package com.jmdoudoux.test.jdom;

import org.jdom.Attribute;
import org.jdom.Document;
import org.jdom.Element;

public class TestJDOM25 extends TestJDOM {

  public static void main(
      String[] args) {

    Element racine = new Element("html");
    Document document = new Document(racine);
    Element body = new Element("body");
    racine.addContent(body);
    Element table = new Element("table");
    body.addContent(table);
    Attribute attribut = new Attribute("width","100%");
    table.setAttribute(attribut);
    body.setAttribute(attribut);

    afficher(document);
  }
}
```

Résultat :

```
Exception in thread "main" org.jdom.IllegalAddException:
The attribute already has an existing parent "table"
        at org.jdom.AttributeList.add(AttributeList.java:187)
        at org.jdom.AttributeList.add(AttributeList.java:131)
        at org.jdom.Element.setAttribute(Element.java:1181)
```

```
        at com.jmdoudoux.test.jdom.TestJDOM25.main(TestJDOM25.java:23)
```

Pour déplacer un attribut vers un autre élément, il faut au préalable utiliser la méthode detach().

Exemple :

```
package com.jmdoudoux.test.jdom;

import org.jdom.Attribute;
import org.jdom.Document;
import org.jdom.Element;

public class TestJDOM26 extends TestJDOM {

  public static void main(
      String[] args) {

    Element racine = new Element("html");
    Document document = new Document(racine);
    Element body = new Element("body");
    racine.addContent(body);
    Element table = new Element("table");
    body.addContent(table);
    Attribute attribut = new Attribute("width","100%");
    table.setAttribute(attribut);
    attribut.detach();
    body.setAttribute(attribut);

    afficher(document);
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<html>
  <body width="100%">
    <table />
  </body>
</html>
```

36.1.5.5. La classe Text

JDOM encapsule un noeud de type texte du document XML dans la classe Text.

Généralement, il n'est pas utile d'utiliser directement cette classe car JDOM propose généralement d'utiliser la classe String sauf lors du parcours des éléments fils retournée par la méthode getContent().

Exemple :

```
package com.jmdoudoux.test.jdom;

import java.util.List;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.output.Format;
import org.jdom.output.XMLOutputter;

public class TestJDOM15 {

  public static void main(
      String[] args) {
```

```
    Element racine = new Element("bibliotheque");
    Document document = new Document(racine);
    Element adresse = new Element("adresse");
    adresse.addContent("mon adresse");
    racine.addContent(adresse);

    List elements = adresse.getContent();
    for (Object element : elements) {
      System.out.println(element.getClass().getName());
    }
  }
}
```

Résultat :

```
org.jdom.Text
```

La classe Texte stocke les caractères dans un champ value auquel il est possible d'accéder via la propriété Text (getText() et setText()).

Il n'est pas utile d'échapper les caractères utilisés par XML(<, >, &, ...). Il suffit de les fournir tel quel et JDOM les échappera lors de l'exportation du document.

Exemple :

```
package com.jmdoudoux.test.jdom;

import org.jdom.Document;
import org.jdom.Element;

public class TestJDOM16 extends TestJDOM {

  public static void main(
      String[] args) {

    Element racine = new Element("bibliotheque");
    Document document = new Document(racine);
    Element adresse = new Element("adresse");
    adresse.addContent("mon adresse < 5 et > 10 & impaire ");
    racine.addContent(adresse);

    afficher(document);
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<bibliotheque>
  <adresse>mon adresse &lt; 5 et &gt; 10 &amp; impaire</adresse>
</bibliotheque>
```

La classe Text possède de nombreuses méthodes

Méthode	Rôle
String getText()	Obtenir la valeur du texte
String getTextTrim()	
String getTextNormalize()	
String normalizeString(String)	

Text setText(String)	Modifier la valeur du texte
void append(String)	Ajouter la chaîne de caractères à la valeur du texte
void append(Text)	Ajoute la valeur du texte de l'objet fourni à la valeur du texte
Element getParent();	Obtenir l'élément qui contient le texte
Document getDocument()	Obtenir le document qui contient le texte
Text setParent(Element)	Associer le texte à son élément parent
Text detach()	Détacher le texte de son élément père

La classe Text possède plusieurs propriétés :

- Value : la valeur du texte
- Parent : l'élément parent
- Document : le document qui contient l'objet

JDOM ne garantit pas que le contenu textuel d'un élément soit stocké dans un unique objet Text.

36.1.5.6. La classe Comment

La classe org.jdom.Comment encapsule un commentaire dans le document.

Exemple :

```
package com.jmdoudoux.test.jdom;

import org.jdom.Comment;
import org.jdom.Document;
import org.jdom.Element;

public class TestJDOM18 extends TestJDOM {

  public static void main(
      String[] args) {

    Element racine = new Element("bibliotheque");
    Comment comment1 = new Comment("mon commentaire bibliotheque");
    racine.addContent(comment1);
    Document document = new Document(racine);
    Element adresse = new Element("adresse");
    Comment comment2 = new Comment("mon commentaire adresse");
    adresse.addContent(comment2);
    adresse.addContent("mon adresse");

    racine.addContent(adresse);

    afficher(document);
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<bibliotheque>
  <!--mon commentaire bibliotheque-->
  <adresse>
    <!--mon commentaire adresse-->
    mon adresse
  </adresse>
</bibliotheque>
```

Il est possible d'ajouter un commentaire directement au document :

- pour ajouter un commentaire à la fin, il suffit d'utiliser la méthode addContent de la classe Document
- pour ajouter un commentaire au début du document (entre le prologue et le tag racine), il faut obtenir la liste des éléments du document grâce à la méthode getContent() et ajouter le commentaire à la première position de la collection

Exemple :

```
package com.jmdoudoux.test.jdom;

import java.util.List;

import org.jdom.Comment;
import org.jdom.Document;
import org.jdom.Element;

public class TestJDOM19 extends TestJDOM {

  public static void main(
      String[] args) {

    Element racine = new Element("bibliotheque");
    Document document = new Document(racine);
    document.addContent(new Comment("mon commentaire de fin"));
    Element adresse = new Element("adresse");
    racine.addContent(adresse);

    Comment comment = new Comment(
    "mon commentaire de debut");
    List content = document.getContent();
    content.add(0, comment);

    afficher(document);
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<!--mon commentaire de debut-->
<bibliotheque>
  <adresse />
</bibliotheque>
<!--mon commentaire de fin-->
```

Lors de l'instanciation d'un objet de type Comment, l'API vérifie que le contenu du commentaire respecte les spécifications XML et lève une exception de type IllegalDataException si celles-ci ne sont pas respectées.

Exemple :

```
package com.jmdoudoux.test.jdom;

import org.jdom.Comment;
import org.jdom.Document;
import org.jdom.Element;

public class TestJDOM20 extends TestJDOM {

  public static void main(
      String[] args) {

    Element racine = new Element("bibliotheque");
    Comment comment1 = new Comment("mon commentaire -- bibliotheque");
    racine.addContent(comment1);
    Document document = new Document(racine);
    Element adresse = new Element("adresse");
    adresse.addContent("mon adresse");
```

```
    racine.addContent(adresse);

    afficher(document);
  }
}
```

Exemple :

```
Exception in thread "main" org.jdom.IllegalDataException:
The data "mon commentaire -- bibliotheque"
is not legal for a JDOM comment: Comments cannot contain double hyphens (--).
        at org.jdom.Comment.setText(Comment.java:120)
        at org.jdom.Comment.<init>(Comment.java:86)
        at com.jmdoudoux.test.jdom.TestJDOM20.main(TestJDOM20.java:13)
```

36.1.5.7. La classe Namespace

La classe org.jdom.Namespace encapsule un espace de nommage associé à un élément ou à un attribut. Chaque Namespace possède un URI. Si l'espace de nommage n'est pas celui par défaut, alors il doit avoir un préfixe sinon le préfixe est une chaîne vide.

Il peut y avoir autant d'espace de nommage que nécessaire dans un même document.

Pour limiter l'occupation mémoire requise par l'espace de nommage de chaque éléments, il n'existe qu'une seule instance de la classe Namespace pour un même espace de nommage. Ceci est garantit par le fait que la classe Namespace ne possède pas de constructeur accessible et fait office de fabrique pour ces instances.

La méthode statique getNamespace() permet de retrouver ou de créer un espace de nom : elle attend en paramètre une URI et/ou un préfixe. Elle permet d'assurer que son appel avec les mêmes paramètres renvoie systématiquement la même instance de la classe Namespace.

Exemple :

```
      Namespace ns1 = Namespace.getNamespace("http://www.jmdoudoux.com");
      Namespace ns2 = Namespace.getNamespace("bibliotheque", "http://www.jmdoudoux.com");
```

Un objet de type Namespace peut être passé en paramètre du constructeur des classes Element et Attribute.

La surcharge de la méthode getNamespace() qui attend uniquement l'uri en paramètre permet de définir un espace de nommage par défaut.

Exemple :

```
package com.jmdoudoux.test.jdom;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.Namespace;

public class TestJDOM71 extends TestJDOM {

  public static void main(
      String[] args) {

    Namespace ns = Namespace.getNamespace("http://www.jmdoudoux.com");

    Element racine = new Element("bibliotheque", ns);
    Document document = new Document(racine);

    Element livre = new Element("livre", ns);
    racine.addContent(livre);
```

```
    livre.addContent(new Element("titre", ns).setText("titre1"));

    afficher(document);
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<bibliotheque xmlns="http://www.jmdoudoux.com">
  <livre>
    <titre>titre1</titre>
  </livre>
</bibliotheque>
```

La surcharge de la méthode getNamespace() qui attend le préfixe et l'uri en paramètre permet de définir un espace de nommage possédant un préfixe.

Exemple :

```
package com.jmdoudoux.test.jdom;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.Namespace;

public class TestJDOM72 extends TestJDOM {

  public static void main(
      String[] args) {

    Namespace ns = Namespace.getNamespace("bibliotheque","http://www.jmdoudoux.com");

    Element racine = new Element("bibliotheque", ns);
    Document document = new Document(racine);

    Element livre = new Element("livre", ns);
    racine.addContent(livre);

    livre.addContent(new Element("titre", ns).setText("titre1"));

    afficher(document);
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<bibliotheque:bibliotheque xmlns:bibliotheque="http://www.jmdoudoux.com">
  <bibliotheque:livre>
    <bibliotheque:titre>titre1</bibliotheque:titre>
  </bibliotheque:livre>
</bibliotheque:bibliotheque>
```

Il n'est pas possible de créer un élément dont le nom contient un caractère « : » : celui-ci est réservé par les spécifications de XML pour préciser un espace de nommage.

Exemple :

```
package com.jmdoudoux.test.jdom;

import org.jdom.Document;
import org.jdom.Element;
```

```
import org.jdom.Namespace;

public class TestJDOM73 extends TestJDOM {

  public static void main(
      String[] args) {

    Namespace ns = Namespace.getNamespace("bibliotheque","http://www.jmdoudoux.com");

    Element racine = new Element("bibliotheque", ns);
    Document document = new Document(racine);

    Element livre = new Element("bibliotheque:livre");
    racine.addContent(livre);

    livre.addContent(new Element("titre", ns).setText("titre1"));

    afficher(document);
  }
}
```

Résultat :

```
Exception in thread "main" org.jdom.IllegalNameException: The name "bibliotheque:livre"
is not legal for JDOM/XML elements: Element names cannot contain colons.
        at org.jdom.Element.setName(Element.java:207)
        at org.jdom.Element.<init>(Element.java:141)
        at org.jdom.Element.<init>(Element.java:153)
        at com.jmdoudoux.test.jdom.TestJDOM73.main(TestJDOM73.java:17)
```

Il faut utiliser une des surcharges du constructeur de la classe Element pour fournir l'espace de nommage.

Exemple :

```
package com.jmdoudoux.test.jdom;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.Namespace;

public class TestJDOM74 extends TestJDOM {

  public static void main(
      String[] args) {

    Namespace ns = Namespace.getNamespace("bibliotheque","http://www.jmdoudoux.com");

    Element racine = new Element("bibliotheque", ns);
    Document document = new Document(racine);

    Element livre = new Element("livre", "bibliotheque", "http://www.jmdoudoux.com");
    racine.addContent(livre);

    livre.addContent(new Element("titre", ns).setText("titre1"));

    afficher(document);
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<bibliotheque:bibliotheque xmlns:bibliotheque="http://www.jmdoudoux.com">
  <bibliotheque:livre>
    <bibliotheque:titre>titre1</bibliotheque:titre>
  </bibliotheque:livre>
</bibliotheque:bibliotheque>
```

Comme chaque objet de type Element et Attribute possède une référence sur un objet de type Namespace, il n'y a pas besoin de se préoccuper de l'espace de nommage lors d'un déplacement d'un élément.

Pour obtenir des éléments fils appartenant à un espace de nommage en utilisant les méthodes getChild() et getChildren() de la classe Element, il faut obligatoirement utiliser la surcharge de ces méthodes qui attend en paramètre un objet de type Namespace.

36.1.5.8. La classe CData

La classe CDATA est une sous classe de la classe Text. La grande différence entre ces deux classes est la façon dont leurs données seront exportées par un objet de type XMLOutputter.

Ces données sont incluses dans une section CDATA et les caractères spéciaux qu'elles contiennent ne sont pas échappés.

Exemple :

```
package com.jmdoudoux.test.jdom;

import org.jdom.CDATA;
import org.jdom.Document;
import org.jdom.Element;

public class TestJDOM17 extends TestJDOM {

  public static void main(
      String[] args) {

    Element racine = new Element("bibliotheque");
    Document document = new Document(racine);
    Element adresse = new Element("adresse");
    adresse.addContent("mon adresse < 5 et > 10 & impaire ");
    racine.addContent(adresse);
    CDATA cData = new CDATA("mon adresse < 5 et > 10 & impaire ");
    racine.addContent(cData);

    afficher(document);
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<bibliotheque>
  <adresse>mon adresse &lt; 5 et &gt; 10 &amp; impaire</adresse>
  <![CDATA[mon adresse < 5 et > 10 & impaire]]>
</bibliotheque>
```

36.1.5.9. La classe ProcessingInstruction

La classe ProcessingInstruction encapsule une instruction de traitement du document XML. Ces instructions permettent de fournir des informations aux outils qui traitent le document XML.

Une instruction est composée d'une cible (target) et de données (data).

La classe ProcessingInstruction possède plusieurs propriétés :

- target : le nom de la cible de l'instruction
- data : données de l'instruction

* parent : l'élément qui contient l'instruction
* document : le document qui contient l'instruction

Lors de l'instanciation d'un objet de type ProcessingInstruction, l'API vérifie que le nom de la cible respecte les spécifications XML et lève une exception de type IllegalTargetException si celles-ci ne sont pas respectées.

Exemple :

```
package com.jmdoudoux.test.jdom;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.ProcessingInstruction;

public class TestJDOM14 extends TestJDOM {

  public static void main(
      String[] args) {

    Element racine = new Element("html");
    Document document = new Document(racine);
    Element body = new Element("body");
    ProcessingInstruction pi = new ProcessingInstruction("php","echo 'bonjour';");
    body.addContent(pi);
    racine.addContent(body);

    afficher(document);
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<html>
  <body>
    <?php echo 'bonjour';?>
  </body>
</html>
```

Comme il est fréquent qu'une instruction de traitement possède des attributs, une surcharge du constructeur attend en paramètre le nom de la cible et un objet de type Map qui encapsule les attributs sous la forme clé/valeur.

Exemple :

```
package com.jmdoudoux.test.jdom;

import java.util.HashMap;
import java.util.Map;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.ProcessingInstruction;

public class TestJDOM45 extends TestJDOM {

  public static void main(
      String[] args) {

    Element racine = new Element("bibliotheque");
    Document document = new Document(racine);

    Map instructions = new HashMap();
    instructions.put("href", "bibliotheque.xsl");
    instructions.put("type", "text/xsl");
    ProcessingInstruction pi = new ProcessingInstruction("xml-stylesheet", instructions);
    document.getContent().add(0, pi);

    Element adresse = new Element("adresse");
```

```
    adresse.addContent("mon adresse");
    racine.addContent(adresse);

    afficher(document);
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="application/xml" href="bibliotheque.xsl"?>
<bibliotheque>
  <adresse>mon adresse</adresse>
</bibliotheque>
```

La méthode getParent() permet de savoir à quel Element est associé l'instruction. Cette méthode renvoie null si l'instruction est rattachée directement au document ou si elle n'est pas encore attachée.

La classe ProcessingInstruction propose la méthode getData() qui renvoie toutes les données sous la forme d'une chaîne de caractères.

Fréquemment les données sont sous la forme d'attributs donc la classe ProcessingInstrution propose des méthodes pour faciliter leur manipulation :

- la méthode getPseudoAttributeValue() qui attend en paramètre le nom de l'attribut renvoie sa valeur
- la méthode getPseudoAttributeNames() renvoie une collection des noms d'attributs

Pour obtenir une instruction de traitement d'un élément, il faut utiliser la méthode getContent() de la classe Element ou Document et parcourir la collection pour obtenir celles de type ProcessingInstruction.

Exemple :

```
package com.jmdoudoux.test.jdom;

import java.io.IOException;
import java.io.StringReader;
import java.util.Iterator;
import java.util.List;

import org.jdom.Document;
import org.jdom.JDOMException;
import org.jdom.ProcessingInstruction;
import org.jdom.input.SAXBuilder;

public class TestJDOM45 extends TestJDOM {

  public static void main(
      String[] args) {

    StringBuilder sb = new StringBuilder("<?xml version=\"1.0\" encoding=\"UTF-8\"?>");
    sb.append("<?xml-stylesheet type=\"text/xsl\" href=\"bibliotheque.xsl\"?>");
    sb.append("<bibliotheque>");
    sb.append("  <adresse>mon adresse</adresse>");
    sb.append("</bibliotheque>");

    SAXBuilder builder = new SAXBuilder();
    Document document;
    try {
      document = builder.build(new StringReader(sb.toString()));

      List elements = document.getContent();
      Iterator iterator = elements.iterator();
      while (iterator.hasNext()) {
        Object o = iterator.next();
        if (o instanceof ProcessingInstruction) {
          ProcessingInstruction pi = (ProcessingInstruction) o;
          System.out.println("PI : "+pi.getTarget());
```

```
          List names = pi.getPseudoAttributeNames();
          Iterator itNames = names.iterator();
          while (itNames.hasNext()) {
            String name = itNames.next().toString();
            System.out.println("    "+name+ " = "+pi.getPseudoAttributeValue(name));
          }
        }
      }

    } catch (JDOMException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
PI : xml-stylesheet
   type = text/xsl
   href = bibliotheque.xsl
```

36.1.6. La création d'un document

JDOM encapsule un document XML dans une arborescence d'objets dont le point d'entrée est un objet de type org.jdom.Document.

La création d'un nouveau document se fait simplement en instanciant un objet de type Document grâce à l'opérateur new sur un des constructeurs de la classe.

Une instance de la classe Document peut aussi facilement être créée à partir d'un document XML existant en utilisant les classes SAXBuilder ou DOMBuilder du package org.jdom.input.

36.1.6.1. La création d'un nouveau document

Le plus simple pour créer un nouveau document est d'instancier un objet de type Element qui va encapsuler la racine du document et de passer cette instance au constructeur de la classe Document.

Exemple :

```
package com.jmdoudoux.test.jdom;

import org.jdom.Document;
import org.jdom.Element;

public class TestJDOM48 {

  public static void main(
      String[] args) {

    Element racine = new Element("bibliotheque");
    Document document = new Document(racine);

  }
}
```

JDOM est beaucoup plus simple que DOM : la création d'un nouveau document avec Dom est plus verbeux.

Exemple :

```
package com.jmdoudoux.test.jdom;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;

import org.w3c.dom.DOMImplementation;
import org.w3c.dom.Document;
import org.w3c.dom.Element;

public class TestJDOM47 {

  public static void main(
      String[] args) {

    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
    DocumentBuilder builder;
    try {
      builder = factory.newDocumentBuilder();
      DOMImplementation impl = builder.getDOMImplementation();
      Document doc = impl.createDocument(null, null, null);
      Element element = doc.createElement("bibliotheque");
    } catch (ParserConfigurationException e) {
      e.printStackTrace();
    }

  }
}
```

Il est aussi possible d'instancier un objet de type Document avec le constructeur par défaut et de lui associer l'élément racine en utilisant la méthode setRootElement().

Exemple :

```
    Document document = new Document();
    Element racine = new Element("bibliotheque");
    document.setRootElement(racine) ;
```

Dans ce cas, le document débute son existence dans un état illégal : il n'est pas possible de faire des opérations sur le document tant que sa racine n'est pas précisée sinon une exception de type IllegalStateException est levée.

36.1.6.2. L'obtention d'une instance de Document à partir d'un document XML

Pour obtenir un objet de type Document à partir d'un document XML existant, JDOM propose deux classes regroupées dans le package org.jdom.input qui implémentent l'interface Builder.

Cette interface définit la méthode build() qui renvoie un objet de type Document et qui est surchargée pour utiliser plusieurs sources différentes : flux (InputStream, Reader) , fichier (File) et URL (URL et String).

Les deux classes sont SAXBuilder et DOMBuilder.

JDOM ne fournit aucun parseur XML : il utilise celui précisé via JAXP ou celui explicitement demandé.

36.1.6.2.1. L'obtention d'une instance de Document à partir d'un document XML en utilisant SAX

La classe SAXBuilder permet d'analyser le document XML avec un parseur de type SAX compatible JAXP, de créer un arbre JDOM et renvoie un objet de type Document.

La mise en oeuvre de la classe SAXBuilder est très simple et nécessite simplement deux étapes :

* Instanciation d'un objet de type SAXBuilder en utilisant un de ses constructeurs
* Invocation de la méthode build() en lui passant en paramètre la ressource permettant l'accès au document

Si l'opération réussie, la méthode build() retourne un objet de type Document encapsulant le document sinon elle lève une exception de type JDOMException si l'analyse du document échoue ou une exception liée à l'accès au document en cas de besoin (par exemple une exception de type IOException).

Exemple :

```
package com.jmdoudoux.test.jdom;

import org.jdom.*;
import org.jdom.input.*;
import java.io.*;

public class TestJDOM3 {
  public static void main(
      String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      Document document = builder.build(new File("test.xml"));
    } catch (JDOMException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

La méthode build() peut lever une exception de type IOException en cas de problème lors de la lecture du document ou une exception de type JDOMException si le document n'est pas correctement formé.

L'exception JDOMException est la classe mère de plusieurs exceptions notamment JDOMParseException qui permet de signifier un problème dans les traitements de JDOM.

Exemple avec le document XML :

```
<?xml version="1.0" encoding="UTF-8"?>
<html>
  <head></head>
  <body>
    <table width="100%" border="0"></table>
    <test>
  </body>
</html>
```

Résultat :

```
org.jdom.input.JDOMParseException: Error on line 7 of document
file:/C:/Documents%20and%20Settings/jmd/workspace/TestJDOM/test.xml:
The element type "test" must be terminated by the matching end-tag "</test>".
      at org.jdom.input.SAXBuilder.build(SAXBuilder.java:501)
      at org.jdom.input.SAXBuilder.build(SAXBuilder.java:847)
      at org.jdom.input.SAXBuilder.build(SAXBuilder.java:826)
      at com.jmdoudoux.test.jdom.TestJDOM3.main(TestJDOM3.java:10)
Caused by: org.xml.sax.SAXParseException: The element type "test"
must be terminated by the matching end-tag "</test>".
      at com.sun.org.apache.xerces.internal.util.ErrorHandlerWrapper.createSAXParseException
      (ErrorHandlerWrapper.java:195)
      at com.sun.org.apache.xerces.internal.util.ErrorHandlerWrapper.fatalError
      (ErrorHandlerWrapper.java:174)
      at com.sun.org.apache.xerces.internal.impl.XMLErrorReporter.reportError
      (XMLErrorReporter.java:388)
      at com.sun.org.apache.xerces.internal.impl.XMLScanner.reportFatalError
      (XMLScanner.java:1411)
      at com.sun.org.apache.xerces.internal.impl.XMLDocumentFragmentScannerImpl.scanEndElement
      (XMLDocumentFragmentScannerImpl.java:1739)
      at com.sun.org.apache.xerces.internal.impl.XMLDocumentFragmentScannerImpl
```

```
        $FragmentContentDriver.next(XMLDocumentFragmentScannerImpl.java:2923)
        at com.sun.org.apache.xerces.internal.impl.XMLDocumentScannerImpl.next
        (XMLDocumentScannerImpl.java:645)
        at com.sun.org.apache.xerces.internal.impl.XMLNSDocumentScannerImpl.next
        (XMLNSDocumentScannerImpl.java:140)
        at com.sun.org.apache.xerces.internal.impl.XMLDocumentFragmentScannerImpl.scanDocument
        (XMLDocumentFragmentScannerImpl.java:508)
        at com.sun.org.apache.xerces.internal.parsers.XML11Configuration.parse
        (XML11Configuration.java:807)
        at com.sun.org.apache.xerces.internal.parsers.XML11Configuration.parse
        (XML11Configuration.java:737)
        at com.sun.org.apache.xerces.internal.parsers.XMLParser.parse(XMLParser.java:107)
        at com.sun.org.apache.xerces.internal.parsers.AbstractSAXParser.parse
        (AbstractSAXParser.java:1205)
        at com.sun.org.apache.xerces.internal.jaxp.SAXParserImpl$JAXPSAXParser.parse
        (SAXParserImpl.java:522)
        at org.jdom.input.SAXBuilder.build(SAXBuilder.java:489)
        ... 3 more
```

Par défaut, le parseur utilisé par JDOM est celui précisé par JAXP ou à défaut Xerces.

SAXBuilder possède plusieurs constructeurs qui permettent de préciser la classe du parseur (nom de la classe pleinement qualifiée de type XMLReader) à utiliser et/ou un booléen qui indique si le document doit être validé.

Par défaut, les vérifications faites par SAXBuilder sur le document XML ne concernent que le fait que le document soit bien formé. Pour valider le document explicitement, il faut demander la validation en passant la valeur true au paramètre validate du constructeur de SAXBuilder.

Exemple :

```
package com.jmdoudoux.test.jdom;
import org.jdom.*;
import org.jdom.input.*;
import java.io.*;

public class TestJDOM3 {
  public static void main(String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder(true);
      Document document = builder.build(new File("test.xml"));
    } catch(JDOMException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
org.jdom.input.JDOMParseException: Error on line 2 of document
file:/C:/Documents%20and%20Settings/jmd/workspace/TestJDOM/test.xml:
Document is invalid: no grammar found.
        at org.jdom.input.SAXBuilder.build(SAXBuilder.java:501)
        at org.jdom.input.SAXBuilder.build(SAXBuilder.java:847)
        at org.jdom.input.SAXBuilder.build(SAXBuilder.java:826)
        at com.jmdoudoux.test.jdom.TestJDOM3.main(TestJDOM3.java:10)
Caused by: org.xml.sax.SAXParseException: Document is invalid: no grammar found.
        at com.sun.org.apache.xerces.internal.util.ErrorHandlerWrapper.createSAXParseException
        (ErrorHandlerWrapper.java:195)
        at com.sun.org.apache.xerces.internal.util.ErrorHandlerWrapper.error
        (ErrorHandlerWrapper.java:131)
        at com.sun.org.apache.xerces.internal.impl.XMLErrorReporter.reportError
        (XMLErrorReporter.java:384)
        at com.sun.org.apache.xerces.internal.impl.XMLErrorReporter.reportError
        (XMLErrorReporter.java:318)
...
```

Dans l'exemple ci-dessus, la validation échoue car aucune DTD n'est précisée dans le document.

Il est possible de configurer le parseur SAX utilisé par SAXBuilder grâce à plusieurs méthodes :

- setErrorHandler(ErrorHandler)
- setEntityResolver(EntityResolver)
- setDTDHandler(DTDHandler)
- setIgnoringElementContentWhitespace(boolean)
- setFeature(String *name*, boolean *value*)
- setProperty(String *name*, Object *value*)

Il est possible de construire un arbre JDOM en utilisant la classe SAXBuilder à partir d'une chaîne de caractères contenant le document XML. Il suffit d'instancier un objet de type StringReader() en lui passant en paramètre la chaîne de caractères contenant le document.

Exemple :

```
package com.jmdoudoux.test.jdom;

import java.io.IOException;
import java.io.StringReader;

import org.jdom.Document;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;

public class TestJDOM49 {
  public static void main(
      String[] args) {
    try {
      StringBuilder sb = new StringBuilder("<?xml version=\"1.0\" encoding=\"UTF-8\"?>");
      sb.append("<?xml-stylesheet type=\"text/xsl\" href=\"bibliotheque.xsl\"?>");
      sb.append("<bibliotheque>");
      sb.append("  <adresse>mon adresse complete</adresse>");
      sb.append("</bibliotheque>");

      SAXBuilder builder = new SAXBuilder();
      Document document = builder.build(new StringReader(sb.toString()));
    } catch (JDOMException e) {
      e.printStackTrace(System.out);
    } catch (IOException e) {
      e.printStackTrace(System.out);
    }
  }
}
```

L'utilisation de SAXBuilder est particulièrement recommandée car elle consomme moins de ressources.

36.1.6.2.2. L'obtention d'une instance de Document à partir d'un arbre DOM

Bien qu'ayant des similitudes de nom, JDOM n'est pas compatible avec DOM. Il n'est pas possible d'utiliser des éléments d'une API dans l'autre directement.

JDOM propose cependant de convertir un arbre DOM en modèle JDOM et vice et versa. Ceci est pratique pour intégrer JDOM dans du code existant manipulant des arbres DOM.

Pour créer un modèle JDOM à partir d'un arbre DOM, il faut utiliser la classe org.jdom.input.DomBuilder. Sa mise en oeuvre est similaire à celle de la classe SAXBuidler : instancier un objet de type DOMBuilder et invoquer sa méthode build().

La classe DOMBuilder propose deux surcharges de la méthode build() :

- org.jdom.Document build(org.w3c.dom.Document) : créer un document JDOM à partir d'un document DOM

* org.jdom.Element build(org.w3c.dom.Element) : créer un élément JDOM à partir d'un élment DOM

Les modifications faites dans le modèle JDOM n'ont aucun impacts sur l'arbre DOM utilisé initialement pour créer l'arbre d'objets JDOM et vice et versa.

36.1.6.3. La création d'éléments

Pour créer un nouvel élément, il suffit d'instancier un objet de la classe Element. Tous les constructeurs de cette classe attendent au moins en paramètre le nom de l'élément. Ce nom doit respecter les spécifications XML, sinon une exception de type IllegalNameException est levée.

L'exception IllegalNameException hérite de l'exception IllegalArgumentException qui est une exception de type runtime : il n'est donc pas obligatoire de traiter ce type d'exception mais elle peut tout de même être levée à l'exécution.

Exemple :

```
package com.jmdoudoux.test.jdom;

import org.jdom.Document;
import org.jdom.Element;

public class TestJDOM29 extends TestJDOM {

  public static void main(
      String[] args) {

    Element racine = new Element("bibliotheque");
    Document document = new Document(racine);
    Element livres = new Element("*livres");
    racine.addContent(livres);

    afficher(document);
  }
}
```

Résultat :

```
Exception in thread "main" org.jdom.IllegalNameException:
The name "*livres" is not legal for JDOM/XML elements:
XML names cannot begin with the character "*".
        at org.jdom.Element.setName(Element.java:207)
        at org.jdom.Element.<init>(Element.java:141)
        at org.jdom.Element.<init>(Element.java:153)
        at com.jmdoudoux.test.jdom.TestJDOM29.main(TestJDOM29.java:13)
```

Chaque Element peut avoir autant d'objets de type Element fils que nécessaire pour représenter le document XML.

Il est possible de préciser le texte associé à l'élément en utilisant la méthode setText() de la classe Element.

36.1.6.4. L'ajout d'éléments fils

Pour ajouter un élément fils à un élément, il faut instancier l'élément fils et utiliser la méthode addContent() de l'instance de l'objet père.

Exemple :

```
package com.jmdoudoux.test.jdom;

import org.jdom.Document;
import org.jdom.Element;

public class TestJDOM30 extends TestJDOM {
```

```
  public static void main(
      String[] args) {

    Element racine = new Element("bibliotheque");
    Document document = new Document(racine);
    Element livres = new Element("livres");
    racine.addContent(livres);

    Element livre = new Element("livre");
    livres.addContent(livre);

    Element titre = new Element("titre").setText("Titre livre 1");
    Element auteur = new Element("auteur").setText("Auteur 1");
    livre.addContent(titre);
    livre.addContent(auteur);

    afficher(document);
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<bibliotheque>
  <livres>
    <livre>
      <titre>Titre livre 1</titre>
      <auteur>Auteur 1</auteur>
    </livre>
  </livres>
</bibliotheque>
```

Comme la plupart des méthodes qui modifie un Element renvoie l'élément lui-même, il est possible de chaîner les différents appels aux méthodes.

Exemple :

```
package com.jmdoudoux.test.jdom;

import org.jdom.Document;
import org.jdom.Element;

public class TestJDOM28 extends TestJDOM {

  public static void main(
      String[] args) {

    Element racine = new Element("bibliotheque");
    Document document = new Document(racine);
    Element livres = new Element("livres");
    racine.addContent(livres);
    livres.addContent(new Element("livre")
    .addContent(new Element("titre").setText("Titre livre 1"))
    .addContent(new Element("auteur").setText("Auteur 1"))
    );

    afficher(document);
  }
}
```

Attention : la lisibilité du code devient moins triviale.

Pour ses propres besoins, il est possible de définir sa propre classe qui hérite de la classe Element pour par exemple créer une portion de document.

Exemple :

```
package com.jmdoudoux.test.jdom;

import org.jdom.Element;

public class ElementLivre extends Element {

  private static final long serialVersionUID = 1L;

  public ElementLivre(String titre, String auteur) {
    super("Livre");
    addContent(new Element("titre").setText(titre));
    addContent(new Element("auteur").setText(auteur));
  }
}
```

Il suffit alors d'utiliser cette classe pour créer le document.

Exemple :

```
package com.jmdoudoux.test.jdom;

import org.jdom.Document;
import org.jdom.Element;

public class TestJDOM31 extends TestJDOM {

  public static void main(
      String[] args) {

    Element racine = new Element("bibliotheque");
    Document document = new Document(racine);
    Element livres = new Element("livres");
    racine.addContent(livres);

    Element livre = new ElementLivre("Titre livre 1","Auteur 1");
    livres.addContent(livre);

    afficher(document);
  }
}
```

36.1.7. L'arborescence d'éléments

Un document XML possède une structure arborescente dans laquelle un Element peut avoir des Elements fils.

Cette section va utiliser le document xml suivant

Exemple : le fichier bibliotheque.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<bibliotheque>
  <livre>
    <!-- commentaires livre 1 -->
    <titre>titre1</titre>
    <auteur>auteur1</auteur>
    <editeur>editeur1</editeur>
  </livre>
    <livre>
    <!-- commentaires livre 2 -->
    <titre>titre2</titre>
    <auteur>auteur2</auteur>
    <editeur>editeur2</editeur>
  </livre>
    <livre>
    <!-- commentaires livre 3 -->
```

```
    <titre>titre3</titre>
    <auteur>auteur3</auteur>
    <editeur>editeur3</editeur>
  </livre>
</bibliotheque>
```

36.1.7.1. Le parcours des éléments

Le parcours des éléments avec JDOM utilise le framework Collection notamment les classes List et Iterator.

La méthode getChildren() de la classe Element renvoie une collection qui encapsule les éléments fils uniquement de premier niveau. Elle possède plusieurs surcharges :

Méthode	Rôle
List getChildren()	Renvoyer tous les éléments fils de premier niveau
List getChildren(String)	Renvoyer tous les éléments fils de premier niveau dont le nom est fourni en paramètre
List getChildren(String, Namescape)	Renvoyer tous les éléments fils de premier niveau dont le nom et l'espace de nommage sont fournis en paramètres

Il suffit alors de réaliser une itération sur la collection pour effectuer les traitements voulus.

Exemple :

```
package com.jmdoudoux.test.jdom;

import java.io.File;
import java.io.IOException;
import java.util.List;
import java.util.ListIterator;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;

public class TestJDOM5 {
  public static void main(
      String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      Document document = builder.build(new File("bibliotheque.xml"));

      Element element = document.getRootElement();
      List livres = element.getChildren("livre");
      ListIterator iterator = livres.listIterator();
      while (iterator.hasNext()) {
        Element el = (Element) iterator.next();
        System.out.println("titre = " + el.getChild("titre").getText());
      }
    } catch (JDOMException e) {
      e.printStackTrace(System.out);
    } catch (IOException e) {
      e.printStackTrace(System.out);
    }
  }
}
```

Résultat :

```
titre = titre1
```

```
titre = titre2
titre = titre3
```

L'inconvénient d'utiliser un nom de tag pour rechercher un élément est qu'il faut être sûr que l'élément existe sinon une exception de type NullPointerException est levée.

Le plus simple est d'avoir une DTD et d'activer la validation du document par le parseur.

Attention : le nom des éléments utilisés est sensible à la casse.

JDOM n'utilise pas encore les generics : il est donc nécessaire de réaliser des casts et des tests de type sur les éléments des collections.

36.1.7.2. L'accès direct à un élément fils

Plutôt que d'itérer sur les éléments fils, il est plus rapide d'utiliser la méthode getChild() de la classe Element surtout si il n'y qu'un seul élément fils ou que c'est le premier que l'on souhaite obtenir.

La méthode getChild() possède deux surcharges :

Méthode	Rôle
Element getChild(String)	Renvoyer le premier fils dont le nom est fourni en paramètre
Element getChild(String, Namespace)	Renvoyer le premier fils dont le nom et l'espace de nommage sont fournis en paramètres

Si aucun fils ne correspond alors la méthode renvoie null.

Si plusieurs fils correspondent alors la méthode getChild() renvoie uniquement le premier.

Exemple :

```
package com.jmdoudoux.test.jdom;

import java.io.File;
import java.io.IOException;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;

public class TestJDOM41 {
  public static void main(
      String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      builder.setIgnoringElementContentWhitespace(true);
      Document document = builder.build(new File("bibliotheque.xml"));

      Element elementRacine = document.getRootElement();
      Element elementLivre = elementRacine.getChild("livre");
      Element elementAuteur = elementLivre.getChild("auteur");
      System.out.println("auteur du premier livre = "+elementAuteur.getText());

    } catch (JDOMException e) {
      e.printStackTrace(System.out);
    } catch (IOException e) {
      e.printStackTrace(System.out);
    }
  }
}
```

Résultat :

```
auteur du premier livre = auteur1
```

Il est possible de chaîner les appels à la méthode getChild() puisqu'elle renvoie l'élément correspondant.

Exemple :

```
package com.jmdoudoux.test.jdom;

import java.io.File;
import java.io.IOException;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;

public class TestJDOM42 {
  public static void main(
      String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      builder.setIgnoringElementContentWhitespace(true);
      Document document = builder.build(new File("bibliotheque.xml"));

      Element elementAuteur = document.getRootElement().getChild("livre").getChild("auteur");
      System.out.println("auteur du premier livre = " + elementAuteur.getText());

    } catch (JDOMException e) {
      e.printStackTrace(System.out);
    } catch (IOException e) {
      e.printStackTrace(System.out);
    }
  }
}
```

La méthode getChild() ne renvoie que le premier fils correspond au critère fourni : si plusieurs éléments fils répondent au critère, il faut utiliser la méthode getChildren() et itérer sur la collection.

Si aucun élément fils ne répond au critère alors la méthode getChild() renvoie null ce qui impliquera la levée d'une exception de type NullPointerException.

Pour éviter ceci, il est nécessaire de connaître la structure du document XML et que celui-ci soit validé via une DTD ou un schéma.

Attention, si un élément père contient la définition d'un espace de nommage, il est nécessaire de le préciser.

Exemple : le document XML utilisé

```
<?xml version="1.0" encoding="UTF-8"?>
<bibliotheque xmlns="http://www.jmdoudoux.com">>
  <livre>
    <!-- commentaires livre 1 -->
    <titre>titre1</titre>
    <auteur>auteur1</auteur>
    <editeur>editeur1</editeur>
  </livre>
    <livre>
    <!-- commentaires livre 2 -->
    <titre>titre2</titre>
    <auteur>auteur2</auteur>
    <editeur>editeur2</editeur>
  </livre>
    <livre>
    <!-- commentaires livre 3 -->
```

```
    <titre>titre3</titre>
    <auteur>auteur3</auteur>
    <editeur>editeur3</editeur>
  </livre>
</bibliotheque>
```

Dans cette version du document XML, la balise racine définit un espace de nommage qui est donc propagé à tous ces éléments fils.

Exemple :

```
package com.jmdoudoux.test.jdom;

import java.io.File;
import java.io.IOException;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;

public class TestJDOM44 {
  public static void main(
      String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      builder.setIgnoringElementContentWhitespace(true);
      Document document = builder.build(new File("bibliotheque.xml"));

      Element elementRacine = document.getRootElement();
      Element elementLivre = elementRacine.getChild("livre");
      Element elementAuteur = elementLivre.getChild("auteur");
      System.out.println("auteur du premier livre = "+elementAuteur.getText());

    } catch (JDOMException e) {
      e.printStackTrace(System.out);
    } catch (IOException e) {
      e.printStackTrace(System.out);
    }
  }
}
```

Résultat :

```
Exception in thread "main" java.lang.NullPointerException
        at com.jmdoudoux.test.jdom.TestJDOM44.main(TestJDOM44.java:21)
```

Dans cet exemple, l'objet elementLivre est null car la méthode getChild() renvoie null : il n'existe pas dans le document d'élément fils de l'élément racine avec le nom « livre » et l'espace de nommage par défaut.

Pour que l'exemple précédent fonctionne, il est nécessaire de préciser l'espace de nommage en paramètre de la méthode getChild()

Exemple :

```
package com.jmdoudoux.test.jdom;

import java.io.File;
import java.io.IOException;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.JDOMException;
import org.jdom.Namespace;
import org.jdom.input.SAXBuilder;

public class TestJDOM44 {
```

```
  public static void main(
      String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      builder.setIgnoringElementContentWhitespace(true);
      Document document = builder.build(new File("bibliotheque.xml"));

      Namespace ns = Namespace.getNamespace("http://www.jmdoudoux.com");

      Element elementRacine = document.getRootElement();
      Element elementLivre = elementRacine.getChild("livre", ns);
      Element elementAuteur = elementLivre.getChild("auteur", ns);
      System.out.println("auteur du premier livre = "+elementAuteur.getText());

    } catch (JDOMException e) {
      e.printStackTrace(System.out);
    } catch (IOException e) {
      e.printStackTrace(System.out);
    }
  }
}
```

Il est aussi possible via l'API Collection d'accéder directement à un des éléments fils.

Exemple :

```
package com.jmdoudoux.test.jdom;

import java.io.File;
import java.io.IOException;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;

public class TestJDOM43 {
  public static void main(
      String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      builder.setIgnoringElementContentWhitespace(true);
      Document document = builder.build(new File("bibliotheque.xml"));

      Element elementLivre = (Element) document.getRootElement().getChildren("livre").get(2);
      Element elementAuteur = elementLivre.getChild("auteur");
      System.out.println("auteur du troisieme livre = " + elementAuteur.getText());

    } catch (JDOMException e) {
      e.printStackTrace(System.out);
    } catch (IOException e) {
      e.printStackTrace(System.out);
    }
  }
}
```

Résultat :

```
auteur du troisieme livre = auteur3
```

36.1.7.3. Le parcours de toute l'arborescence d'un document

JDOM ne fournit aucune API permettant facilement de parcourir tout le document comme peut le proposer DOM. Il faut utiliser une méthode récursive.

Exemple :

```
package com.jmdoudoux.test.jdom;

import java.io.File;
import java.io.IOException;
import java.util.Arrays;
import java.util.Iterator;
import java.util.List;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;

public class TestJDOM32 {
  public static void main(
      String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      builder.setIgnoringElementContentWhitespace(true);
      Document document = builder.build(new File("bibliotheque.xml"));

      Element elementRacine = document.getRootElement();
      afficherFils(elementRacine, 0);
    } catch (JDOMException e) {
      e.printStackTrace(System.out);
    } catch (IOException e) {
      e.printStackTrace(System.out);
    }
  }

  /**
   * Methode recursive qui parcours un element et affiche ces fils
   * @param element
   * @param niveau
   */
  private static void afficherFils(Element element, int niveau) {
    String indentation = getIndentation(niveau);
    StringBuilder ligne = new StringBuilder(indentation);

    ligne.append(element.getName());
    if(element.getChildren().isEmpty()) {
      ligne.append(" = ");
      ligne.append(element.getText());
    }

    System.out.println(ligne.toString());

    List fils = element.getChildren();
    Iterator iterator = fils.iterator();
    while (iterator.hasNext()) {
      Element elementFils = (Element) iterator.next();
      afficherFils(elementFils, niveau+1);
    }

  }

  private static String getIndentation(int n) {
    char[] car = new char[n];
    Arrays.fill(car, 0, n, ' ');
    return new String(car);
  }

}
```

Résultat :

```
bibliotheque
 livre
  titre = titre1
  auteur = auteur1
  editeur = editeur1
```

```
 livre
  titre = titre2
  auteur = auteur2
  editeur = editeur2
 livre
  titre = titre3
  auteur = auteur3
  editeur = editeur3
```

La méthode getChildren() ne renvoie que des Elements. Pour obtenir les autres entités liées à l'élément, il faut utiliser la méthode getContent() qui renvoie toutes les entités (commentaires, texte, ...) y compris les éléments fils sous la forme d'une collection.

Pour traiter chaque élément de cette collection, il est nécessaire de faire un test de type sur l'occurrence de la collection en cours de traitement grâce à l'opérateur instanceof.

Exemple :

```
package com.jmdoudoux.test.jdom;

import java.io.File;
import java.io.IOException;
import java.util.Arrays;
import java.util.Iterator;
import java.util.List;

import org.jdom.Comment;
import org.jdom.Document;
import org.jdom.Element;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;

public class TestJDOM33 {
  public static void main(
      String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      builder.setIgnoringElementContentWhitespace(true);
      Document document = builder.build(new File("bibliotheque.xml"));

      Element elementRacine = document.getRootElement();
      afficherFils(elementRacine, 0);
    } catch (JDOMException e) {
      e.printStackTrace(System.out);
    } catch (IOException e) {
      e.printStackTrace(System.out);
    }
  }

  /**
   * Methode recursive qui parcours un element et affiche ces fils
   * @param element
   * @param niveau
   */
  private static void afficherFils(Element element, int niveau) {
    String indentation = getIndentation(niveau);
    StringBuilder ligne = new StringBuilder(indentation);

    ligne.append(element.getName());
    if(element.getChildren().isEmpty()) {
      ligne.append(" = ");
      ligne.append(element.getText());
    }

    System.out.println(ligne.toString());

    List fils = element.getContent();
    Iterator iterator = fils.iterator();
    while (iterator.hasNext()) {
      Object objetFils = iterator.next();
```

```
      if (objetFils instanceof Element) {
        Element elementFils = (Element) objetFils;
        afficherFils(elementFils, niveau+1);
      } else {
        if (objetFils instanceof Comment) {
          Comment com = (Comment) objetFils;
          System.out.println(indentation+" -- "+com.getValue());
        }
      }
    }

  }

  private static String getIndentation(int n) {
    char[] car = new char[n];
    Arrays.fill(car, 0, n, ' ');
    return new String(car);
  }
}
```

Résultat :

```
bibliotheque
 livre
  --  commentaires livre 1
  titre = titre1
  auteur = auteur1
  editeur = editeur1
 livre
  --  commentaires livre 2
  titre = titre2
  auteur = auteur2
  editeur = editeur2
 livre
  --  commentaires livre 3
  titre = titre3
  auteur = auteur3
  editeur = editeur3
```

36.1.7.4. Les éléments parents

JDOM permet une navigation descendante de l'arbre des objets mais aussi ascendante.

Chaque élément à un unique élément père sauf l'élément racine qui n'en a pas.

La méthode getParent() renvoie l'élément parent de l'élément courant. Elle renvoie null pour l'élément racine du document.

Il est ainsi possible de remonter récursivement dans l'arborescence de niveau en niveau jusqu'à ce qu'il n'y ait plus d'élément parent.

Remarque : un élément qui n'est pas encore rattaché à un document ne possède pas non plus d'élément père : dans ce cas les méthodes getParent() et getDocument() renvoient null.

La classe Element propose la méthode isRootElement() qui renvoie true si l'élément est la racine du document.

La classe Element propose aussi la méthode isAncestor() qui renvoie un booléen indiquant si l'élément est un ancêtre de l'élément fourni en paramètre.

36.1.8. La modification d'un document

La classe Element propose de nombreuses méthodes pour modifier un élément :

Méthode	Rôle
addContent()	Ajouter l'entité fournie en paramètre en tant que fils de l'élément. Plusieurs surcharges existent pour des paramètres de type Content, Collection ou String et certaines permettent de définir l'index
removeAttribute()	Supprimer un attribut de l'élément
removeChild()	Supprimer un élément fils
removeChildren()	Supprimer tous les éléments fils dont le nom est fourni en paramètre
removeContent()	Supprimer une entité fille
setAttribute()	Créer ou modifier un attribut
setContent()	Remplacer un noeud ou tous les noeuds de l'élément
setName()	Modifier le nom de l'élément
setText()	Modifier le texte de l'élément

Exemple :

```
package com.jmdoudoux.test.jdom;

import java.io.File;
import java.io.IOException;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;

public class TestJDOM66 extends TestJDOM {
  public static void main(String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      Document document = builder.build(new File("bibliotheque.xml"));

      Element livre2 = (Element) document.getRootElement().getChildren().get(1);

      // Ajouter un nouvel élément
      livre2.addContent(new Element("publication").setText("1996"));
      // Supprimer tous les noeuds nommé editeur
      livre2.removeChildren("editeur");

      afficher(document);
    } catch(JDOMException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<bibliotheque>
  <livre>
    <!-- commentaires livre 1 -->
    <titre>titre1</titre>
    <auteur>auteur1</auteur>
    <editeur>editeur1</editeur>
  </livre>
  <livre>
    <!-- commentaires livre 2 -->
```

```
    <titre>titre2</titre>
    <auteur>auteur2</auteur>
    <publication>1996</publication>
  </livre>
  <livre>
    <!-- commentaires livre 3 -->
    <titre>titre3</titre>
    <auteur>auteur3</auteur>
    <editeur>editeur3</editeur>
  </livre>
</bibliotheque>
```

L'appel à la méthode getChildren() renvoie une collection des noeuds fils de l'élément courant. Cette collection est dynamique et peut être directement modifiée pour ajouter/enlever des noeuds, réordonner les noeuds simplement en utilisant les méthodes de l'API Collection.

Exemple :

```
package com.jmdoudoux.test.jdom;

import java.io.File;
import java.io.IOException;
import java.util.List;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;

public class TestJDOM65 extends TestJDOM {
  public static void main(String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      Document document = builder.build(new File("bibliotheque.xml"));

      Element livre2 = (Element) document.getRootElement().getChildren().get(1);

      List fils = livre2.getChildren();

      // Ajouter un nouvel élément
      fils.add(new Element("publication").setText("1996"));
      // Aajouter un nouveau noeud après le second noeud
      fils.add(1, new Element("isbn").setText("0000000000"));


      Element livre3 = (Element) document.getRootElement().getChildren().get(2);
      fils = livre3.getChildren();
      // supprimer le second noeud
      fils.remove(1);
      // Supprimer tous les neouds nommé editeur
      fils.removeAll(livre3.getChildren("editeur"));

      afficher(document);
    } catch(JDOMException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<bibliotheque>
  <livre>
    <!-- commentaires livre 1 -->
    <titre>titre1</titre>
    <auteur>auteur1</auteur>
    <editeur>editeur1</editeur>
```

```
  </livre>
  <livre>
    <!-- commentaires livre 2 -->
    <titre>titre2</titre>
    <isbn>0000000000</isbn>
    <auteur>auteur2</auteur>
    <editeur>editeur2</editeur>
    <publication>1996</publication>
  </livre>
  <livre>
    <!-- commentaires livre 3 -->
    <titre>titre3</titre>
  </livre>
</bibliotheque>
```

L'utilisation de l'API collection implique de tenir compte des contraintes qu'elle impose. Par exemple, il faut être vigilent aux modifications de la collection lors de son parcours grâce à un itérateur.

Exemple :

```
package com.jmdoudoux.test.jdom;

import java.io.File;
import java.io.IOException;
import java.util.Iterator;
import java.util.List;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;

public class TestJDOM69 extends TestJDOM {
  public static void main(String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      Document document = builder.build(new File("bibliotheque.xml"));

      Element racine = document.getRootElement();
      Element livre2 = (Element) racine.getChildren().get(1);

      List livre2Fils = livre2.getChildren();
      Iterator itr = livre2Fils.iterator();
      while (itr.hasNext()) {
        Element fils = (Element) itr.next();
        if ("auteur".equals(fils.getName())) {
          fils.detach();
        }
      }

      afficher(document);
    } catch(JDOMException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
Exception in thread "main" java.util.ConcurrentModificationException
   at org.jdom.ContentList$FilterListIterator.checkConcurrentModification(ContentList.java:940)
   at org.jdom.ContentList$FilterListIterator.nextIndex(ContentList.java:829)
   at org.jdom.ContentList$FilterListIterator.hasNext(ContentList.java:785)
   at com.jmdoudoux.test.jdom.TestJDOM69.main(TestJDOM69.java:23)
```

Une exception de type ConcurrentModificationException est levée car la méthode detach() modifie le contenu de la

collection de façon concurrente au parcours fait par l'itérateur. Par palier à ce problème, il ne faut pas utiliser la méthode detach() de la classe Element mais utiliser la méthode remove() de l'itérateur.

Exemple :

```
package com.jmdoudoux.test.jdom;

import java.io.File;
import java.io.IOException;
import java.util.Iterator;
import java.util.List;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;

public class TestJDOM69 extends TestJDOM {
  public static void main(String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      Document document = builder.build(new File("bibliotheque.xml"));

      Element racine = document.getRootElement();
      Element livre2 = (Element) racine.getChildren().get(1);

      List livre2Fils = livre2.getChildren();
      Iterator itr = livre2Fils.iterator();
      while (itr.hasNext()) {
        Element fils = (Element) itr.next();
        if ("auteur".equals(fils.getName())) {
          itr.remove();
        }
      }

      afficher(document);
    } catch(JDOMException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<bibliotheque>
  <livre>
    <!-- commentaires livre 1 -->
    <titre>titre1</titre>
    <auteur>auteur1</auteur>
    <editeur>editeur1</editeur>
  </livre>
  <livre>
    <!-- commentaires livre 2 -->
    <titre>titre2</titre>
    <editeur>editeur2</editeur>
  </livre>
  <livre>
    <!-- commentaires livre 3 -->
    <titre>titre3</titre>
    <auteur>auteur3</auteur>
    <editeur>editeur3</editeur>
  </livre>
</bibliotheque>
```

36.1.8.1. L'obtention du texte d'un élément

La classe Element propose plusieurs méthodes pour obtenir et modifier le texte d'un élément

Méthode	Rôle
String getText()	Renvoyer le texte de l'élément
String getTextTrim()	Renvoyer le texte de l'élément sans les espaces de début et de fin
String getTextNormalize()	Renvoyer le texte de l'élément sans les espaces de début et de fin et tous les espaces consécutifs sont remplacés par un espace unique
Element setText(String)	Forcer la création d'un noeud unique de type texte

Exemple :

```
package com.jmdoudoux.test.jdom;

import org.jdom.Element;

public class TestJDOM51 extends TestJDOM {

  public static void main(
      String[] args) {

    Element texte = new Element("texte");
    texte.setText("  mon   texte avec     des   espaces   ");
    System.out.println("getText()=*"+texte.getText()+"*");
    System.out.println("getTextTrim()=*"+texte.getTextTrim()+"*");
    System.out.println("getTextNormalize()=*"+texte.getTextNormalize()+"*");
  }
}
```

Résultat :

```
getText()=*  mon   texte avec     des   espaces   *
getTextTrim()=*mon   texte avec     des   espaces*
getTextNormalize()=*mon texte avec des espaces*
```

Dans le cas où un commentaire ou une instruction de traitement est inclus dans le texte, celui ci est ignoré lors de l'appel à la méthode getText().

Exemple :

```
package com.jmdoudoux.test.jdom;

import org.jdom.Element;
import java.io.IOException;
import java.io.StringReader;

import org.jdom.Document;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;

public class TestJDOM63 extends TestJDOM {

  public static void main(
      String[] args) {

    try {
      StringBuilder sb = new StringBuilder("<?xml version=\"1.0\" encoding=\"UTF-8\"?>");
      sb.append("<?xml-stylesheet type=\"text/xsl\" href=\"bibliotheque.xsl\"?>");
      sb.append("<bibliotheque>");
      sb.append("  <adresse>mon adresse <!-- commentaire -->complete</adresse>");
      sb.append("</bibliotheque>");

      SAXBuilder builder = new SAXBuilder();
      Document document = builder.build(new StringReader(sb.toString()));
```

```
      Element adresse = document.getRootElement().getChild("adresse");
      System.out.println("getText()=*" + adresse.getText() + "*");
      System.out.println("getTextTrim()=*" + adresse.getTextTrim() + "*");
      System.out.println("getTextNormalize()=*" + adresse.getTextNormalize() + "*");

    } catch (JDOMException e) {
      e.printStackTrace(System.out);
    } catch (IOException e) {
      e.printStackTrace(System.out);
    }
  }
}
```

Résultat :

```
getText()=*mon adresse complete*
getTextTrim()=*mon adresse complete*
getTextNormalize()=*mon adresse complete*
```

De plus, si le texte contient des éléments fils, le texte de ces derniers n'est pas repris par l'appel à la méthode getText().

Exemple :

```
package com.jmdoudoux.test.jdom;

import org.jdom.Element;
import java.io.IOException;
import java.io.StringReader;

import org.jdom.Document;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;

public class TestJDOM64 extends TestJDOM {

  public static void main(
      String[] args) {

    try {
      StringBuilder sb = new StringBuilder("<?xml version=\"1.0\" encoding=\"UTF-8\"?>");
      sb.append("<bibliotheque>");
      sb.append("  <adresse>mon adresse <b>complete</b> et integrale</adresse>");
      sb.append("</bibliotheque>");

      SAXBuilder builder = new SAXBuilder();
      Document document = builder.build(new StringReader(sb.toString()));

      Element adresse = document.getRootElement().getChild("adresse");
      System.out.println("getText()=*" + adresse.getText() + "*");
      System.out.println("getTextTrim()=*" + adresse.getTextTrim() + "*");
      System.out.println("getTextNormalize()=*" + adresse.getTextNormalize() + "*");

    } catch (JDOMException e) {
      e.printStackTrace(System.out);
    } catch (IOException e) {
      e.printStackTrace(System.out);
    }
  }
}
```

Résultat :

```
getText()=*mon adresse  et integrale*
getTextTrim()=*mon adresse  et integrale*
getTextNormalize()=*mon adresse et integrale*
```

Pour obtenir l'intégralité du texte incluant le texte des éléments fils, il est nécessaire d'écrire un morceau de code qui va parcourir le noeud et ces éléments fils en concaténant le résultat des appels à l'appel de chaque méthode getText() des noeuds.

36.1.8.2. La modification du texte d'un élément

La méthode setText() permet de modifier le texte d'un élément. Attention, son utilisation supprime tous les noeuds de l'élément déjà existant.

Exemple :

```
package com.jmdoudoux.test.jdom;

import org.jdom.Comment;
import org.jdom.Document;
import org.jdom.Element;

public class TestJDOM52 extends TestJDOM {

  public static void main(
      String[] args) {

    Element racine = new Element("racine");
    Document document = new Document(racine);
    Element texte = new Element("test");
    racine.addContent(texte);
    Element fils = new Element("fils");
    texte.addContent(fils);
    Comment commentaire = new Comment("mon commentaire");
    texte.addContent(commentaire);
    texte.setText("mon texte");

    afficher(document);
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<racine>
  <test>mon texte</test>
</racine>
```

Pour éviter la suppression des noeuds fils, il faut utiliser la méthode addContent() plutôt que la méthode setText().

Exemple :

```
package com.jmdoudoux.test.jdom;

import org.jdom.Comment;
import org.jdom.Document;
import org.jdom.Element;

public class TestJDOM53 extends TestJDOM {

  public static void main(
      String[] args) {

    Element racine = new Element("racine");
    Document document = new Document(racine);
    Element texte = new Element("test");
    racine.addContent(texte);
    Element fils = new Element("fils");
    texte.addContent(fils);
    Comment commentaire = new Comment("mon commentaire");
    texte.addContent(commentaire);
```

```
    texte.addContent("mon texte");

    afficher(document);
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<racine>
  <test>
    <fils />
    <!--mon commentaire-->
    mon texte
  </test>
</racine>
```

Il ne faut pas utiliser de séquences d'échappement dans le texte d'un élément. JDOM prend le texte tel quel et les caractères seront échappés lors de l'exportation du document.

Exemple :

```
package com.jmdoudoux.test.jdom;

import org.jdom.Document;
import org.jdom.Element;

public class TestJDOM54 extends TestJDOM {

  public static void main(
      String[] args) {

    Element racine = new Element("racine");
    Document document = new Document(racine);
    Element texte = new Element("texte");
    racine.addContent(texte);
    Element test1 = new Element("test1");
    test1.setText("&#002A;");
    texte.addContent(test1);
    Element test2 = new Element("test2");
    test2.setText("\u002A");
    texte.addContent(test2);

    afficher(document);
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<racine>
  <texte>
    <test1>&amp;#002A;</test1>
    <test2>*</test2>
  </texte>
</racine>
```

36.1.8.3. L'obtention du texte d'un élément fils

Il est fréquent dans un document de vouloir obtenir le texte d'un élément fils.

La classe Element propose plusieurs méthodes pour obtenir facilement le texte d'un élément fils. Ces méthodes possèdent deux surcharges : une attendant en paramètre le nom du tag fils, l'autre le nom du tag fils et son espace de nommage.

Méthode	Rôle
String getChildText()	Renvoyer le texte du premier élément fils
String getChildTextTrim()	Renvoyer le texte du premier élément fils sans les espaces de début et de fin
String getChildTextNormalize()	Renvoyer le texte du premier élément fils sans les espaces de début et de fin et tous les espaces consécutifs sont remplacés par un espace unique

Exemple :

```
package com.jmdoudoux.test.jdom;

import java.io.File;
import java.io.IOException;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.JDOMException;
import org.jdom.Namespace;
import org.jdom.input.SAXBuilder;

public class TestJDOM55 {
  public static void main(
      String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      builder.setIgnoringElementContentWhitespace(true);
      Document document = builder.build(new File("bibliotheque.xml"));

      Namespace ns = Namespace.getNamespace("http://www.jmdoudoux.com");

      Element elementRacine = document.getRootElement();
      Element elementLivre = elementRacine.getChild("livre", ns);
      String titre = elementLivre.getChildText("titre", ns);
      System.out.println("titre du premier livre = "+titre);
      String auteur = elementLivre.getChildText("auteur", ns);
      System.out.println("auteur du premier livre = "+auteur);
      String editeur = elementLivre.getChildText("editeur", ns);
      System.out.println("editeur du premier livre = "+editeur);

    } catch (JDOMException e) {
      e.printStackTrace(System.out);
    } catch (IOException e) {
      e.printStackTrace(System.out);
    }
  }
}
```

Résultat :

```
titre du premier livre = titre1
auteur du premier livre = auteur1
editeur du premier livre = editeur1
```

Il est préférable de valider le document XML utilisé avant d'utiliser ces méthodes :

- Elles ne renvoient que le premier élément fils répondant au critère
- Si aucun élément n'est trouvé alors elles renvoient null

Il ne faut les utiliser que pour des éléments fils uniques qui ne contiennent que des noeuds de type #PCDATA

36.1.8.4. L'ajout et la suppression des fils

La classe Element possède plusieurs surcharges de la méthode addContent() pour ajouter des noeuds fils à l'élément.

Méthode	Rôle
addContent(Content)	Ajouter le noeud fourni à la suite des noeuds existants
addContent(int, Content)	Ajouter le noeud fourni à l'index fourni
addContent(Collection)	Ajouter les collections de noeuds fournis à la suite des noeuds existants
addContent(int, Collection)	Ajouter les collections des noeuds fournis à l'index fourni
addContent(String)	Ajouter un noeud de type Text

Toutes ces méthodes peuvent lever une exception de type IllegalAddException et renvoient l'élément lui même.

La classe Element possède plusieurs surcharges de la méthode removeContent(), removeChild() et removeChildren() pour supprimer des noeuds fils de l'élément.

Méthode	Rôle
removeContent()	Supprimer tous les noeuds fils
removeContent(Content)	Supprimer le noeud fils fourni
removeContent(int)	Supprimer le noeud dont l'index est fourni
removeContent(Filter)	Supprimer les noeuds fils correspondant au filtre fourni
removeChild(String)	Supprimer le premier noeud fils dont le nom est fourni
removeChild(String, Namespace)	Supprimer le premier noeud fils dont le nom et l'espace de nommage sont fournis
removeChildren(String)	Supprimer les noeuds fils dont le nom est fourni
removeChildren(String, Namespace)	Supprimer les noeuds fils dont le nom et l'espace de nommage sont fournis

La méthode removeChildren() ne supprime que les éléments fils de premier niveau répondant aux critères de nom et d'espace de nommage fournis.

Si aucun espace de nommage n'est fourni, seuls les éléments sans espace de nommage entrent dans les critères. Les autres noeuds fils tels que du texte, des commentaires ou des instructions de traitement ne sont pas supprimés.

La suppression d'un élément supprime l'élément mais aussi toute l'arborescence fille de l'élément.

Il est aussi possible d'obtenir une collection des noeuds fils en utilisant la méthode getContent() et de manipuler le contenu de cette collection en utilisant les méthodes add() et remove() de la collection.

Exemple :

```
package com.jmdoudoux.test.jdom;

import java.io.File;
import java.io.IOException;
import java.util.Iterator;
import java.util.List;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;

public class TestJDOM70 extends TestJDOM {
  public static void main(String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      Document document = builder.build(new File("bibliotheque.xml"));
```

```
      Element racine = document.getRootElement();
      supprimerNoeudsTousParNom(racine, "auteur");

      afficher(document);
    } catch(JDOMException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

  /**
   * Supprime récursivement tous les éléments ayant pour nom celui fourni en paramètres
   * @param element element à traiter
   * @param nom nom des éléments à supprimer
   */
  public static void supprimerNoeudsTousParNom(Element element, String nom) {

    List elements = element.getChildren(nom);
    elements.removeAll(elements);

    // recherche de nouveau des fils puisque la collection à potenciellement été modifiée
    List fils = element.getChildren();
    Iterator iterator = fils.iterator();
    while (iterator.hasNext()) {
      supprimerNoeudsTousParNom((Element) iterator.next(), nom);
    }

  }

}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<bibliotheque>
  <livre>
    <!-- commentaires livre 1 -->
    <titre>titre1</titre>
    <editeur>editeur1</editeur>
  </livre>
  <livre>
    <!-- commentaires livre 2 -->
    <titre>titre2</titre>
    <editeur>editeur2</editeur>
  </livre>
  <livre>
    <!-- commentaires livre 3 -->
    <titre>titre3</titre>
    <editeur>editeur3</editeur>
  </livre>
</bibliotheque>
```

36.1.8.5. Le déplacement d'un ou des éléments

Lors de la création d'une instance de type Element, cet élément n'est pas associé à un document. Dans ce cas, la méthode getDocument() renvoie null.

JDOM interdit cependant qu'un Element soit inclus dans deux documents.

Pour déplacer un élément dans un même document ou dans un autre document, il est nécessaire d'invoquer sa méthode detach() au préalable.

Exemple :

```
package com.jmdoudoux.test.jdom;
```

```
import java.io.File;
import java.io.IOException;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.JDOMException;
import org.jdom.Namespace;
import org.jdom.input.SAXBuilder;

public class TestJDOM56 extends TestJDOM {
  public static void main(
      String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      builder.setIgnoringElementContentWhitespace(true);
      Document documentSource = builder.build(new File("bibliotheque.xml"));

      Namespace ns = Namespace.getNamespace("http://www.jmdoudoux.com");

      Element elementLivre = documentSource.getRootElement().getChild("livre", ns);

      Element elementRacine = new Element("Bibliotheque");
      Document documentCible = new Document(elementRacine);

      elementLivre.detach();
      elementRacine.addContent(elementLivre);

      afficher(documentSource);
      afficher(documentCible);
    } catch (JDOMException e) {
      e.printStackTrace(System.out);
    } catch (IOException e) {
      e.printStackTrace(System.out);
    }
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<bibliotheque xmlns="http://www.jmdoudoux.com">
  <livre>
    <!-- commentaires livre 2 -->
    <titre>titre2</titre>
    <auteur>auteur2</auteur>
    <editeur>editeur2</editeur>
  </livre>
  <livre>
    <!-- commentaires livre 3 -->
    <titre>titre3</titre>
    <auteur>auteur3</auteur>
    <editeur>editeur3</editeur>
  </livre>
</bibliotheque>

<?xml version="1.0" encoding="UTF-8"?>
<Bibliotheque>
  <livre xmlns="http://www.jmdoudoux.com">
    <!-- commentaires livre 1 -->
    <titre>titre1</titre>
    <auteur>auteur1</auteur>
    <editeur>editeur1</editeur>
  </livre>
</Bibliotheque>
```

Le déplacement d'un élément implique le déplacement de ces noeuds fils. Les espaces de nommage sont aussi gérés lors de ce déplacement.

36.1.8.6. La duplication d'un élément

Il arrive parfois d'avoir besoin de dupliquer un élément. Pour cela, il suffit simplement d'utiliser la méthode clone() sur l'instance de l'Element.

L'élément est dupliqué en incluant tous ces noeuds descendants.

Il ne reste plus qu'à ajouter le nouvel élément dans l'arborescence du document.

Exemple :

```
package com.jmdoudoux.test.jdom;

import java.io.File;
import java.io.IOException;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;

public class TestJDOM67 extends TestJDOM {
  public static void main(String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      Document document = builder.build(new File("bibliotheque.xml"));

      Element racine = document.getRootElement();

      Element livre2 = (Element) racine.getChildren().get(1);

      racine.addContent((Element) livre2.clone());

      afficher(document);
    } catch(JDOMException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<bibliotheque>
  <livre>
    <!-- commentaires livre 1 -->
    <titre>titre1</titre>
    <auteur>auteur1</auteur>
    <editeur>editeur1</editeur>
  </livre>
  <livre>
    <!-- commentaires livre 2 -->
    <titre>titre2</titre>
    <auteur>auteur2</auteur>
    <editeur>editeur2</editeur>
  </livre>
  <livre>
    <!-- commentaires livre 3 -->
    <titre>titre3</titre>
    <auteur>auteur3</auteur>
    <editeur>editeur3</editeur>
  </livre>
  <livre>
    <!-- commentaires livre 2 -->
    <titre>titre2</titre>
    <auteur>auteur2</auteur>
    <editeur>editeur2</editeur>
  </livre>
</bibliotheque>
```

La méthode cloneContent() renvoie une collection de noeuds fils dupliqués.

Exemple :

```
package com.jmdoudoux.test.jdom;

import java.io.File;
import java.io.IOException;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;

public class TestJDOM68 extends TestJDOM {
  public static void main(String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      Document document = builder.build(new File("bibliotheque.xml"));

      Element racine = document.getRootElement();

      Element livre2 = (Element) racine.getChildren().get(1);

      racine.addContent(livre2.cloneContent());

      afficher(document);
    } catch(JDOMException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<bibliotheque>
  <livre>
    <!-- commentaires livre 1 -->
    <titre>titre1</titre>
    <auteur>auteur1</auteur>
    <editeur>editeur1</editeur>
  </livre>
  <livre>
    <!-- commentaires livre 2 -->
    <titre>titre2</titre>
    <auteur>auteur2</auteur>
    <editeur>editeur2</editeur>
  </livre>
  <livre>
    <!-- commentaires livre 3 -->
    <titre>titre3</titre>
    <auteur>auteur3</auteur>
    <editeur>editeur3</editeur>
  </livre>
  <!-- commentaires livre 2 -->
  <titre>titre2</titre>
  <auteur>auteur2</auteur>
  <editeur>editeur2</editeur>
</bibliotheque>
```

36.1.9. L'utilisation de filtres

JDOM propose l'interface org.jdom.filter.Filter qui permet de définir un filtre.

L'interface Filter ne définit que la méthode matches() qui attend en paramètre un objet correspondant à un noeud et renvoie un booléen pour préciser si l'objet répond ou non aux critères du filtre.

Ce filtre peut être utilisé par plusieurs méthodes de certaines classes de JDOM pour restreindre leur action sur les entités qui répondent aux critères du filtre.

JDOM propose deux implémentations de l'interface Filter :

- ContentFilter : permet de filtrer sur le type de noeuds
- ElementFilter : permet de filtrer sur le nom et/ou l'espace de nommage des éléments

La classe ContentFilter possède plusieurs constructeurs :

Constructeur	Rôle
ContentFilter()	Filtre acceptant tous les types de noeuds
ContentFilter(int)	Filtre acceptant uniquement les noeuds précisés dans le masque (le masque utilise les constantes définies dans la classe ContentFilter)
ContentFilter(boolean)	Filtre acceptant ou non tous les types de noeuds selon la valeur du paramètre

La classe ContentFilter propose plusieurs méthodes setXXXVisible() qui attendent un paramètre de type booléen qui permet d'inclure ou non les noeuds de type XXX.

Exemple : afficher tous les commentaires du document

```
package com.jmdoudoux.test.jdom;

import java.io.File;
import java.io.IOException;
import java.util.Iterator;
import java.util.List;

import org.jdom.Comment;
import org.jdom.Document;
import org.jdom.Element;
import org.jdom.JDOMException;
import org.jdom.filter.ContentFilter;
import org.jdom.input.SAXBuilder;

public class TestJDOM57 extends TestJDOM {
  public static void main(
      String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      builder.setIgnoringElementContentWhitespace(true);
      Document documentSource = builder.build(new File("bibliotheque.xml"));

      Element elementRacine = documentSource.getRootElement();
      process(elementRacine);

    } catch (JDOMException e) {
      e.printStackTrace(System.out);
    } catch (IOException e) {
      e.printStackTrace(System.out);
    }
  }

  public static void process(Element element) {
    List children = element.getContent();
    Iterator iterator = children.iterator();
    while (iterator.hasNext()) {
```

```
      Object o = iterator.next();
      if (o instanceof Element) {
        Element child = (Element) o;
        afficherCommentaires(child);
        process(child);
      }
    }
  }

  public static void afficherCommentaires(Element element) {
    ContentFilter filtre = new ContentFilter(false);
    filtre.setCommentVisible(true);

    List children = element.getContent(filtre);
    Iterator iterator = children.iterator();
    while (iterator.hasNext()) {
      Comment comment = (Comment) iterator.next();
      System.out.println(comment.getText());
    }
  }
}
```

Résultat :

```
commentaires livre 1
commentaires livre 2
commentaires livre 3
```

Le classe ElementFilter possède plusieurs constructeurs :

Constructeur	Rôle
ElementFilter()	Filtre qui ne renvoie que les noeuds de type Element
ElementFilter(String)	Filtre qui ne renvoie que les éléments dont le nom est fourni
ElementFilter(Namespace)	Filtre qui ne renvoie que les éléments dont l'espace de nommage est fourni
ElementFilter(String, Namespace)	Filtre qui ne renvoie que les éléments dont le nom et l'espace de nommage sont fournis

Exemple : afficher le titre de tous les livres

```
package com.jmdoudoux.test.jdom;

import java.io.File;
import java.io.IOException;
import java.util.Iterator;
import java.util.List;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.JDOMException;
import org.jdom.filter.ElementFilter;
import org.jdom.input.SAXBuilder;

public class TestJDOM58 extends TestJDOM {
  public static void main(
      String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      builder.setIgnoringElementContentWhitespace(true);
      Document documentSource = builder.build(new File("bibliotheque.xml"));

      Element elementRacine = documentSource.getRootElement();
      Iterator iterator = elementRacine.getContent().iterator();
      while (iterator.hasNext()) {
        Object o = iterator.next();
        if (o instanceof Element) {
          Element child = (Element) o;
```

```
          afficherTitre(child);
        }
      }

    } catch (JDOMException e) {
      e.printStackTrace(System.out);
    } catch (IOException e) {
      e.printStackTrace(System.out);
    }
  }

  public static void afficherTitre(Element element) {
    ElementFilter filtre = new ElementFilter("titre");

    List children = element.getContent(filtre);
    Iterator iterator = children.iterator();
    while (iterator.hasNext()) {
      Element fils = (Element) iterator.next();
      System.out.println(fils.getText());
    }
  }
}
```

Résultat :

```
titre1
titre2
titre3
```

Il est aussi possible de définir son propre filtre en créant une classe qui implémente l'interface Filter. Il suffit de définir la méthode matches() en lui faisant renvoyer un booléen indiquant le résultat de l'application des critères de filtre sur l'objet fourni en paramètres.

Exemple : afficher les auteurs de chaque livre

Exemple : afficher les auteurs de chaque livre

```
package com.jmdoudoux.test.jdom;

import java.io.File;
import java.io.IOException;
import java.util.Iterator;
import java.util.List;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.JDOMException;
import org.jdom.filter.Filter;
import org.jdom.input.SAXBuilder;

public class TestJDOM59 extends TestJDOM {
  public static void main(
      String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      builder.setIgnoringElementContentWhitespace(true);
      Document documentSource = builder.build(new File("bibliotheque.xml"));

      Element elementRacine = documentSource.getRootElement();
      Iterator iterator = elementRacine.getContent().iterator();
      while (iterator.hasNext()) {
        Object o = iterator.next();
        if (o instanceof Element) {
          Element child = (Element) o;
          afficherTitre(child);
        }
      }

    } catch (JDOMException e) {
      e.printStackTrace(System.out);
```

```
    } catch (IOException e) {
      e.printStackTrace(System.out);
    }
  }

  public static void afficherTitre(Element element) {
    Filter filtre = new Filter() {
      private static final long serialVersionUID = 1L;

      public boolean matches(
          Object arg0) {
        boolean resultat = false;

        if(arg0 instanceof Element){
          Element element = (Element)arg0;

          resultat = "auteur".equals(element.getName());
        }

        return resultat;
      }

    };

    List children = element.getContent(filtre);
    Iterator iterator = children.iterator();
    while (iterator.hasNext()) {
      Element fils = (Element) iterator.next();
      System.out.println(fils.getText());
    }
  }
}
```

Résultat :

```
auteur1
auteur2
auteur3
```

Cet exemple n'a qu'un intérêt pédagogique puisque la même opération peut être réalisée en utilisant la méthode getChildText(). En pratique les filtres personnalisés sont plus complexes.

36.1.10. L'exportation d'un document

JDOM prévoit plusieurs classes pour permettre d'exporter le document contenu dans un objet de type Document. Cette exportation peut se faire :

- Sous la forme de flux : OutputStream ou Writer
- Vers d'autres API : Event Stream (SAX) ou Document (JDOM)

Les classes nécessaires à ces traitements sont regroupées dans le package org.jdom.output.

36.1.10.1. L'exportation dans un flux

La classe XMLOutputter permet d'envoyer le document XML dans un flux. Il est possible de fournir plusieurs paramètres pour formater la sortie du document.

Cette classe possède plusieurs constructeurs dont les principaux sont :

Constructeur	Rôle

XMLOutputter()	Créer un objet par défaut, sans paramètre de formatage
XMLOutputter(Format)	Créer un objet, en précisant en paramètres les options de formattage

La mise en oeuvre de cette classe est très simple : il suffit d'instancier un objet de type XMLOutputter et d'invoquer sa méthode output().

Exemple : lecture et exportation sur la console

```
package com.jmdoudoux.test.jdom;

import java.io.File;
import java.io.IOException;

import org.jdom.Document;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;
import org.jdom.output.XMLOutputter;

public class TestJDOM34 {
  public static void main(String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      Document document = builder.build(new File("test.xml"));
      XMLOutputter sortie = new XMLOutputter();
      sortie.output(document, System.out);

    } catch(JDOMException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

La classe XMLOuputter possède de nombreuses surcharges de la méthode output() permettant l'exportation dans un flux OutputStream ou Writer de différentes entités (Document, Element, Comment, Text, EntityRef, ProcessingInstruction, DocType, ...)

Exemple :

```
package com.jmdoudoux.test.jdom;

import java.io.IOException;

import org.jdom.Element;
import org.jdom.output.XMLOutputter;

public class TestJDOM40 {
  public static void main(String[] args) {
    try {
      Element racine = new Element("html");
      Element body = new Element("body");
      racine.addContent(body);
      Element titre = new Element("H1");
      titre.setText("titre");
      body.addContent(titre);

      XMLOutputter sortie = new XMLOutputter();
      sortie.output(racine, System.out);

    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
<html><body><H1>titre</H1></body></html>
```

Il est possible de configurer les options de formattage de l'exportation en utilisant un objet de type org.jdom.output.Format.

La classe Format propose trois formats prédéfinis que l'on peut obtenir en invoquant la méthode statique correspondante :

- CompactFormat
- PrettyFormat
- RawFormat

Exemple :

```
package com.jmdoudoux.test.jdom;

import java.io.File;
import java.io.IOException;

import org.jdom.Document;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;
import org.jdom.output.Format;
import org.jdom.output.XMLOutputter;

public class TestJDOM35 {
  public static void main(String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      Document document = builder.build(new File("test.xml"));
      XMLOutputter sortie = new XMLOutputter(Format.getCompactFormat());
      sortie.output(document, System.out);

    } catch(JDOMException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat avec CompactFormat

```
<?xml version="1.0" encoding="UTF-8"?>
<html><head /><body><table width="100%" border="0" /></body></html>
```

Résultat avec PrettyFormat

```
<?xml
version="1.0" encoding="UTF-8"?>
<html>
  <head />
  <body>
    <table width="100%"
border="0" />
  </body>
</html>
```

Résultat avec RawFormat

```
<?xml version="1.0" encoding="UTF-8"?>
<html>
<head />
<body>
<table width="100%" border="0" />
</body>
</html>
```

Pour des besoins plus spécifiques, il est possible d'instancier un objet Format et de le configurer en utilisant ses différentes méthodes.

Par défaut, XMLOutputter utilise l'encodage des caractères en UTF-8. Pour préciser un autre encodage des caractères, il faut utiliser la méthode setEncoding() de l'objet Format utilisé par XMLOutputter.

Le flux utilisé peut être de type OutputStream ou Writer. L'utilisation d'un OutputStream est plus facile car le Writer impose de préciser l'encodage de caractères correspondant à celui déclaré dans le prologue du document.

Exemple :

```
package com.jmdoudoux.test.jdom;

import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStreamWriter;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.output.Format;
import org.jdom.output.XMLOutputter;

public class TestJDOM38 extends TestJDOM {

  public static void main(
      String[] args) {

    Element racine = new Element("bibliotheque");
    Document document = new Document(racine);
    Element livres = new Element("livres");
    racine.addContent(livres);

    Element livre = new Element("livre");
    livres.addContent(livre);

    Element titre = new Element("titre").setText("Titre livre 1");
    Element auteur = new Element("auteur").setText("Auteur 1");
    livre.addContent(titre);
    livre.addContent(auteur);

    Format format = Format.getPrettyFormat();
    format.setEncoding("ISO-8859-1");
    XMLOutputter sortie = new XMLOutputter(format);

    FileOutputStream fos = null;
    try {
      fos = new FileOutputStream("c:/temp/test.xml");
      OutputStreamWriter out = new OutputStreamWriter(fos, "ISO-8859-1");
      sortie.output(document, out);

    } catch (IOException e) {
      e.printStackTrace();
    } finally {
      if (fos != null) {
        try {
          fos.close();
        } catch (IOException e) {
          e.printStackTrace();
        }
      }
    }
  }
}
```

La classe XMLOutputter propose aussi la méthode ouputString() qui exporte différentes entités selon la surcharge utilisée dans une chaîne de caractères.

Exemple :

```
package com.jmdoudoux.test.jdom;

import java.io.File;
import java.io.IOException;

import org.jdom.Document;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;
import org.jdom.output.Format;
import org.jdom.output.XMLOutputter;

public class TestJDOM36 {
  public static void main(String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      Document document = builder.build(new File("test.xml"));
      XMLOutputter sortie = new XMLOutputter(Format.getCompactFormat());
      String docXML = sortie.outputString(document);
      System.out.println(docXML);

    } catch(JDOMException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

La classe XMLOutputter se charge de convertir les caractères utilisés par XML en leurs entités respectives durant l'exportation.

Exemple :

```
package com.jmdoudoux.test.jdom;

import java.io.IOException;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.output.Format;
import org.jdom.output.XMLOutputter;

public class TestJDOM37  {

  public static void main(
      String[] args) {

    Element racine = new Element("personne");
    Document document = new Document(racine);
    Element adresse = new Element("adresse");
    adresse.addContent("mon adresse < 5 et > 10 & impaire ");
    racine.addContent(adresse);
    XMLOutputter sortie = new XMLOutputter(Format.getRawFormat());
    try {
      sortie.output(document, System.out);
    } catch (IOException e) {
      e.printStackTrace();
    }

  }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<personne><adresse>mon adresse &lt; 5 et &gt; 10 &amp; impaire </adresse></personne>
```

36.1.10.2. L'exportation dans un arbre DOM

La classe org.jdom.ouput.DOMOutputter permet d'exporter un document JDOM dans un arbre DOM.

Exemple :

```
package com.jmdoudoux.test.jdom;

import java.io.File;
import java.io.IOException;

import org.jdom.Document;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;
import org.jdom.output.DOMOutputter;

public class TestJDOM39 {
  public static void main(String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      Document document = builder.build(new File("test.xml"));

      DOMOutputter domOutputter = new DOMOutputter();
      org.w3c.dom.Document documentDOM = domOutputter.output(document);

    } catch(JDOMException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Cette exportation est plutôt une conversion de l'arbre d'objets du modèle JDOM vers le modèle DOM.

Les deux arbres d'objets sont indépendants l'un de l'autre : une modification dans l'arbre JDOM après l'exportation doit être faite aussi dans l'arbre DOM ou il est nécessaire de refaire une exportation pour refléter la modification dans l'arbre DOM

36.1.10.3. L'exportation en SAX

La classe SAXOutputter permet de générer des événements SAX à partir d'un document JDOM.

La classe SAXOutputter possède plusieurs constructeurs qui attendent tous un objet de type ContentHandler et certains des objets de type ErrorHandler, DTDHandler, EntityResolver et LexicalHandler.

La méthode output() parcours l'arbre JDOM et émet les événements SAX correspondants qui seront traités par les handlers.

Exemple :

```
package com.jmdoudoux.test.jdom;

import java.io.File;
import java.io.IOException;

import org.jdom.Document;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;
import org.jdom.output.SAXOutputter;
import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;
```

```
public class TestJDOM50 {
  public static void main(
      String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      builder.setIgnoringElementContentWhitespace(true);
      Document document = builder.build(new File("bibliotheque.xml"));

      MonSAXHandler handler = new MonSAXHandler();
      SAXOutputter outputter = new SAXOutputter(handler);
      outputter.setErrorHandler(handler);
      outputter.setDTDHandler(handler);
      outputter.output(document);

    } catch (JDOMException e) {
      e.printStackTrace(System.out);
    } catch (IOException e) {
      e.printStackTrace(System.out);
    }
  }
}

class MonSAXHandler extends DefaultHandler {
  private String tagCourant = "";

  /**
   * Actions a réaliser lors de la detection d'un nouvel element.
   */
  public void startElement(
      String nameSpace,
      String localName,
      String qName,
      Attributes attr) throws SAXException {
    tagCourant = localName;
    System.out.println("debut tag : " + localName);
  }

  /**
   * Actions à réaliser lors de la détection de la fin d'un element.
   */
  public void endElement(
      String nameSpace,
      String localName,
      String qName) throws SAXException {
    tagCourant = "";
    System.out.println("Fin tag " + localName);
  }

  /**
   * Actions à réaliser au début du document.
   */
  public void startDocument() {
    System.out.println("Debut du document");
  }

  /**
   * Actions à réaliser lors de la fin du document XML.
   */
  public void endDocument() {
    System.out.println("Fin du document");
  }

  /**
   * Actions à réaliser sur les données
   */
  public void characters(
      char[] caracteres,
      int debut,
      int longueur) throws SAXException {
    String donnees = new String(caracteres, debut, longueur);
    if (!tagCourant.equals("")) {
      if (!Character.isISOControl(caracteres[debut])) {
        System.out.println(" Element " + tagCourant + ", valeur = *" + donnees + "*");
```

```
      }
    }
  }
}
```

Résultat :

```
Debut du document
debut tag : bibliotheque
debut tag : livre
debut tag : titre
 Element titre, valeur = *titre1*
Fin tag titre
debut tag : auteur
 Element auteur, valeur = *auteur1*
Fin tag auteur
debut tag : editeur
 Element editeur, valeur = *editeur1*
Fin tag editeur
Fin tag livre
debut tag : livre
debut tag : titre
 Element titre, valeur = *titre2*
Fin tag titre
debut tag : auteur
 Element auteur, valeur = *auteur2*
Fin tag auteur
debut tag : editeur
 Element editeur, valeur = *editeur2*
Fin tag editeur
Fin tag livre
debut tag : livre
debut tag : titre
 Element titre, valeur = *titre3*
Fin tag titre
debut tag : auteur
 Element auteur, valeur = *auteur3*
Fin tag auteur
debut tag : editeur
 Element editeur, valeur = *editeur3*
Fin tag editeur
Fin tag livre
Fin tag bibliotheque
Fin du document
```

36.1.11. L'utilisation de XSLT

La classe org.jdom.transform.XSLTransformer est un helper qui facilite la mise en oeuvre de transformations simples. Cette classe utilise l'API TrAX de JAXP. Le moteur de transformation utilisé doit être paramétré via JAXP.

La classe XSLTransformer possède plusieurs constructeurs qui attendent en paramètre une feuille de style XSL.

Elle possède plusieurs surcharges de la méthode transform() qui appliquent la feuille de style à un document ou un ensemble de noeuds.

Exemple :

```
package com.jmdoudoux.test.jdom;

import java.io.File;
import java.io.IOException;

import org.jdom.Document;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;
import org.jdom.transform.XSLTransformer;
```

```
public class TestJDOM62 extends TestJDOM {
  public static void main(
      String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      builder.setIgnoringElementContentWhitespace(true);
      Document documentSource = builder.build(new File("bibliotheque.xml"));

      XSLTransformer transformer = new XSLTransformer("bibliotheque.xsl");
      Document documentCible = transformer.transform(documentSource);

      afficher(documentCible);

    } catch (JDOMException e) {
      e.printStackTrace(System.out);
    } catch (IOException e) {
      e.printStackTrace(System.out);
    }
  }
}
```

Exemple : la feuille de style bibliotheque.xsl

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
  <html>
    <body>
      <h2>bibliotheque</h2>
      <table border="1">
        <tr bgcolor="lightblue">
          <th align="left">Titre</th>
          <th align="left">Auteur</th>
        </tr>
        <xsl:for-each select="bibliotheque/livre">
          <tr>
            <td>
              <xsl:value-of select="titre" />
            </td>
            <td>
              <xsl:value-of select="auteur" />
            </td>
          </tr>
        </xsl:for-each>
      </table>
    </body>
  </html>
</xsl:template>
</xsl:stylesheet>
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<html>
  <body>
    <h2>bibliotheque</h2>
    <table border="1">
      <tr bgcolor="lightblue">
        <th align="left">Titre</th>
        <th align="left">Auteur</th>
      </tr>
      <tr>
        <td>titre1</td>
        <td>auteur1</td>
      </tr>
      <tr>
        <td>titre2</td>
        <td>auteur2</td>
      </tr>
      <tr>
        <td>titre3</td>
        <td>auteur3</td>
```

```
      </tr>
    </table>
  </body>
</html>
```

Les méthodes transform() qui attendent en paramètre un objet de type document retourne un objet JDOM de type Document encapsulant le résultat de la transformation. Ceci est particulièrement adapté lorsqu'un document XML est transformé en un autre document XML.

Pour des besoins plus spécifiques, il est possible d'obtenir une instance de la classe Transformer de JAXP et d'utiliser les classes org.jdom.transform.JDOMSource et JDOMResult comme wrapper respectivement en entrée et en sortie de la transformation.

Ceci est nécessaire par exemple lorsque des paramètres doivent être passés à la feuille de style.

Exemple :

```
package com.jmdoudoux.test.jdom;

import java.io.File;
import java.io.IOException;

import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerException;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.stream.StreamSource;

import org.jdom.Document;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;
import org.jdom.transform.JDOMResult;
import org.jdom.transform.JDOMSource;

public class TestJDOM75 extends TestJDOM {
  public static void main(
      String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      builder.setIgnoringElementContentWhitespace(true);
      Document documentSource = builder.build(new File("bibliotheque.xml"));

      TransformerFactory factory = TransformerFactory.newInstance();
      Transformer transformer = factory.newTransformer(new StreamSource("biblio.xsl"));

      JDOMSource source = new JDOMSource(documentSource);
      JDOMResult resultat = new JDOMResult();

      transformer.setParameter("libelle", "mon libelle");
      transformer.transform(source, resultat);

      afficher(resultat.getDocument());

    } catch (JDOMException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    } catch (TransformerException e) {
      e.printStackTrace();
    }
  }
}
```

Exemple : la feuille de style qui déclare et utilise un paramètre nommé libelle

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:param name="libelle" select=""/>
<xsl:template match="/">
  <html>
    <body>
```

```
        <h2>bibliotheque : <xsl:value-of select="$libelle"/> </h2>
        <table border="1">
          <tr bgcolor="lightblue">
            <th align="left">Titre</th>
            <th align="left">Auteur</th>
          </tr>
          <xsl:for-each select="bibliotheque/livre">
            <tr>
              <td>
                <xsl:value-of select="titre" />
              </td>
              <td>
                <xsl:value-of select="auteur" />
              </td>
            </tr>
          </xsl:for-each>
        </table>
      </body>
    </html>
</xsl:template>
</xsl:stylesheet>
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<html>
  <body>
    <h2>bibliotheque : mon libelle</h2>
    <table border="1">
      <tr bgcolor="lightblue">
        <th align="left">Titre</th>
        <th align="left">Auteur</th>
      </tr>
      <tr>
        <td>titre1</td>
        <td>auteur1</td>
      </tr>
      <tr>
        <td>titre2</td>
        <td>auteur2</td>
      </tr>
      <tr>
        <td>titre3</td>
        <td>auteur3</td>
      </tr>
    </table>
  </body>
</html>
```

36.1.12. L'utilisation de XPath

JDOM propose un support de XPath en utilisant Jaxen depuis sa version beta 9.

Les bibliothèques de Jaxen doivent être ajoutées au classpath : elles sont fournies dans le sous répertoire lib de l'archive binaire de JDOM.

Elles peuvent aussi être téléchargées sur le site http://jaxen.org/

Il faut ajouter au classpath les bibliothèques : jaxen-core.jar, jaxen-jdom.jar et saxpath.jar.

Il faut instancier un objet de type org.jdom.xpath.XPath en utilisant sa méthode statique newInstance() qui attend en paramètre l'expression XPath.

L'invocation de la méthode selectNodes() sur cette instance en lui passant en paramètre le document renvoie une collection des noeuds qui répond à l'expression XPath.

Exemple :

```
package com.jmdoudoux.test.jdom;

import java.io.File;
import java.io.IOException;
import java.util.Iterator;
import java.util.List;

import org.jdom.Document;
import org.jdom.Element;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;
import org.jdom.xpath.XPath;

public class TestJDOM60 extends TestJDOM {
  public static void main(
      String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      builder.setIgnoringElementContentWhitespace(true);
      Document document = builder.build(new File("bibliotheque.xml"));

      XPath x      = XPath.newInstance("/bibliotheque/livre");
      List list    = x.selectNodes(document);

      System.out.println("nb livres="+list.size());

      Iterator iterator = list.iterator();
      while (iterator.hasNext()) {
        Object o = iterator.next();
        if (o instanceof Element) {
          Element livre = (Element) o;
          System.out.println("livre : "+livre.getChildText("titre"));
        }
      }

    } catch (JDOMException e) {
      e.printStackTrace(System.out);
    } catch (IOException e) {
      e.printStackTrace(System.out);
    }
  }
}
```

Résultat :

```
nb livres=3
livre : titre1
livre : titre2
livre : titre3
```

Le type des objets contenus dans la collection retournée par la méthode dépend de l'expression XPath fournie.

Exemple :

```
package com.jmdoudoux.test.jdom;

import java.io.File;
import java.io.IOException;
import java.util.Iterator;
import java.util.List;

import org.jdom.Document;
import org.jdom.JDOMException;
import org.jdom.Text;
import org.jdom.input.SAXBuilder;
import org.jdom.xpath.XPath;

public class TestJDOM61 {
  public static void main(
```

```
      String[] args) {
    try {
      SAXBuilder builder = new SAXBuilder();
      builder.setIgnoringElementContentWhitespace(true);
      Document document = builder.build(new File("bibliotheque.xml"));

      XPath x      = XPath.newInstance("/bibliotheque/livre/titre/text()");
      List list    = x.selectNodes(document);

      System.out.println("nb titres="+list.size());

      Iterator iterator = list.iterator();
      while (iterator.hasNext()) {
        Text texte  = (Text) iterator.next();
        System.out.println("livre : "+texte.getValue());
      }

    } catch (JDOMException e) {
      e.printStackTrace(System.out);
    } catch (IOException e) {
      e.printStackTrace(System.out);
    }
  }
}
```

Résultat :

```
nb titres=3
livre : titre1
livre : titre2
livre : titre3
```

La méthode selectSingleNode() peut être utilisée à la place de la méthode selectNodes() lorsque l'expression ne revoie qu'un seul noeud.

36.1.13. L'intégration à Java

JDOM est une API développée en Java pour Java : elle repose sur des API de Java tel que l'API collection et met en oeuvre certaines fonctionnalités de Java notamment le clonage ou la sérialisation.

JDOM a délibérément été voulu non thread safe puisque cela devrait être la plus large utilisation de l'API. Pour une utilisation des objets dans un contexte multi thread, il faut procéder manuellement à une synchronisation des portions de code critiques.

Les méthodes equals() sont redéfinies pour ne retourner l'égalité que si les deux objets sont exactement les mêmes (test sur les références effectué par l'opérateur ==).

Ce test d'égalité permet de garantir que l'élément cible est bien celui concerné notamment en tenant compte de sa position dans le document. Il peut par exemple, y avoir plusieurs éléments avec le même nom et la même valeur dans un même document. Le fait d'avoir des instances distinctes garantit de manipuler l'élément concerné.

De plus, les méthodes equals() et hashCode() sont déclarées final pour ne pas permettre de déroger à cette règle de comparaison.

Les classes qui encapsulent des données du document implémentent l'interface Serializable (sauf la classe Namespace) ainsi ces objets peuvent être sérialisés pour les rendre persistants ou permettre leur échange au travers le réseau.

Les classes qui encapsulent des données du document redéfinissent la méthode toString() pour retourner une représentation textuelle des données qu'elles encapsulent entre crochets en précisant le type de l'entité.

Attention : la méthode toString() ne renvoie pas de représentation au format XML de l'entité. Pour obtenir cette représentation, il faut utiliser un objet de type XMLOutputter.

Les classes qui encapsulent des données du document implémentent l'interface Cloneable sauf la classe Namespace qui encapsule un objet immuable. Le clonage d'un élément se fait de façon récursive : seul le parent de l'objet original n'est pas repris dans la copie.

L'utilisation de l'API Collection pour encapsuler un ensemble d'entités implique que toutes les modifications sur une collection effectue une modification directe sur le document.

De nombreuses méthodes de l'API JDOM peuvent lever une exception qui hérite de la classe JDOMException. Ceci permet de faire un traitement générique sur ces exceptions ou de faire un traitement sur une exception en particulier.

36.1.14. Les contraintes de la mise en oeuvre de JDOM

La mise en oeuvre de JDOM possède plusieurs contraintes dont il faut tenir compte.

L'utilisation de JDOM implique la création en mémoire de l'arbre d'objets encapsulant le document ce qui peut être difficile dans un environnement ayant peu de ressources notamment mémoire ou pour traiter de gros documents.

JDOM ne propose pas de support complet pour les DTD ou les schémas : il n'est pas possible de s'assurer que le document est valide lors d'une modification.

36.2. dom4j

dom4j est un framework open source pour manipuler des données XML, XSL et Xpath. Il est entièrement développé en Java et pour Java.

Dom4j n'est pas un parser mais propose un modèle de représentation d'un document XML et une API pour en faciliter l'utilisation. Pour obtenir une telle représentation, dom4j utilise soit SAX, soit DOM. Comme il est compatible JAXP, il est possible d'utiliser toute implémentation de parser qui implémente cette API.

La version de dom4j utilisée dans cette section est la 1.3

36.2.1. L'installation de dom4j

Il faut télécharger la dernière version à l'url http://dom4j.org/download.html

Il suffit de décompresser le fichier downloadé. Celui-ci contient de nombreuses bibliothèques (ant, xalan, xerces, crimson, junit, ...), le code source du projet et la documentation.

Le plus simple pour utiliser rapidement dom4j est de copier les fichiers jar contenus dans le répertoire lib dans le répertoire ext du répertoire %JAVA_HOME%/jre/lib/ext ainsi que les fichiers dom4j.jar et dom4j-full.jar.

36.2.2. La création d'un document

Dom4j encapsule un document dans un objet de type org.dom4j.Document. Dom4j propose des API pour facilement créer un tel objet qui va être le point d'entrée de la représentation d'un document XML .

Exemple utilisant SAX :

```
import org.dom4j.*;
import org.dom4j.io.*;
```

```
public class Testdom4j_1 {
  public static void main(String args[]) {
    DOCUMENT DOCUMENT;
    try {
      SAXReader xmlReader = new SAXReader();
      document = xmlReader.read("test.xml");
    } catch (Exception e){
      e.printStackTrace();
    }
  }
}
```

Pour exécuter ce code, il suffit d'exécuter

java -cp .;%JAVA_HOME%/jre/lib/ext/dom4j-full.jar Testdom4j_1.bat

Exemple à partir d'une chaîne de caractères :

```
import org.dom4j.*;
public class Testdom4j_8 {
  public static void main(String args[]) {
    Document document = null;
    String texte = "<bibliotheque><livre><titre>titre 1</titre><auteur>auteur 1</auteur>"
         +"<editeur>editeur 1</editeur></livre></bibliotheque>";
    try {
      document = DocumentHelper.parseText(texte);
    } catch (Exception e){
     e.printStackTrace();
    }
  }
}
```

36.2.3. Le parcours d'un document

Le parcours du document construit peut se faire de plusieurs façons :

- utilisation de l'API collection
- utilisation de XPath
- utilisation du pattern Visitor

Le parcours peut se faire en utilisant l'API collection de Java.

Exemple : obtenir tous les noeuds fils du noeud racine

```
import org.dom4j.*;
import org.dom4j.io.*;
import java.util.*;
public class Testdom4j_2 {
  public static void main(String args[]) {
    Document document;
    org.dom4j.Element racine;
    try {
      SAXReader xmlReader = new SAXReader();
      document = xmlReader.read("test.xml");
      racine = document.getRootElement();
      Iterator it = racine.elementIterator();
      while(it.hasNext()){
        Element element = (Element)it.next();
        System.out.println(element.getName());
      }
    } catch (Exception e){
      e.printStackTrace();
    }
  }
}
```

Un des grands intérêts de dom4j est de proposer une recherche dans le document en utilisant la technologie Xpath.

Exemple : obtenir tous les noeuds fils du noeud racine

```
import org.dom4j.*;
import org.dom4j.io.*;
import java.util.*;
public class Testdom4j_3 {
  public static void main(String args[]) {
    Document document;
    try {
      SAXReader xmlReader = new SAXReader();
      document = xmlReader.read("test.xml");
      XPath xpathSelector = DocumentHelper.createXPath("/bibliotheque/livre/auteur");
      List liste = xpathSelector.selectNodes(document);
      for ( Iterator it = liste.iterator(); it.hasNext(); ) {
        Element element = (Element) it.next();
        System.out.println(element.getName()+" : "+element.getText());
      }
    } catch (Exception e){
      e.printStackTrace();
    }
  }
}
```

36.2.4. La modification d'un document XML

L'interface Document propose plusieurs méthodes pour modifier la structure du document.

Méthode	Rôle
Document addComment(String)	Ajouter un commentaire
void setDocType(DocumentType)	Modifier les caractéristiques du type de document
void setRootElement(Element)	Modifier l'élément racine du document

L'interface Element propose plusieurs méthodes pour modifier un élément du document.

Méthode	Rôle
void add(...)	Méthode surchargée qui permet d'ajouter un attribut, une entité, un espace nommage ou du texte à l'élément
Element addAttribute(String, String)	Ajouter un attribut à l'élément
Element addComment(String)	Ajouter un commentaire à l'élément
Element addEntity(String, String)	Ajouter une entité à l'élément
Element addNameSpace(String, String)	Ajouter un espace de nommage à l'élément
Element addText(String)	Ajouter un text à l'élément

36.2.5. La création d'un nouveau document XML

Il est très facile de créer un document XML

La classe DocumentHelper propose une méthode createDocument() qui renvoie une nouvelle instance de la classe Document. Il suffit alors d'ajouter chacun des noeuds de l'arbre du nouveau document XML en utilisant la méthode

addElement() de la classe Document.

Exemple :

```
import org.dom4j.*;
public class Testdom4j_4 {
  public static void main(String args[]) {
    Document document = DocumentHelper.createDocument();
    Element root = document.addElement( "bibliotheque" );
    Element livre = null;
    try {
    livre = root.addElement("livre");
    livre.addElement("titre").addText("titre 1");
    livre.addElement("auteur").addText("auteur 1");
    livre.addElement("editeur").addText("editeur 1");
    } catch (Exception e){
      e.printStackTrace();
    }
  }
}
```

36.2.6. L'exportation d'un document

Pour écrire le document XML dans un fichier, une méthode de la classe Document permet de réaliser cette action très simplement

Exemple :

```
import org.dom4j.*;
import java.io.*;
public class Testdom4j_5 {
  public static void main(String args[]) {
    Document document = DocumentHelper.createDocument();
    Element root = document.addElement( "bibliotheque" );
    Element livre = null;
    try {
      livre = root.addElement("livre");
      livre.addElement("titre").addText("titre 1");
      livre.addElement("auteur").addText("auteur 1");
      livre.addElement("editeur").addText("editeur 1");
      livre = root.addElement("livre");
      livre.addElement("titre").addText("titre 2");
      livre.addElement("auteur").addText("auteur 2");
      livre.addElement("editeur").addText("editeur 2");
      livre = root.addElement("livre");
      livre.addElement("titre").addText("titre 3");
      livre.addElement("auteur").addText("auteur 3");
      livre.addElement("editeur").addText("editeur 3");
      FileWriter out = new FileWriter( "test2.xml" );
      document.write( out );
      out.close();
    } catch (Exception e){
      e.printStackTrace();
    }
  }
}
```

Pour pouvoir agir sur le formatage du document ou pour utiliser un flux différent, il faut utiliser la classe XMLWriter

Exemple :

```
import org.dom4j.*;
import org.dom4j.io.*;
import java.io.*;
public class Testdom4j_6 {
```

```
  public static void main(String args[]) {
    Document document = DocumentHelper.createDocument();
    Element root = document.addElement( "bibliotheque" );
    Element livre = null;
    try {
      livre = root.addElement("livre");
      livre.addElement("titre").addText("titre 1");
      livre.addElement("auteur").addText("auteur 1");
      livre.addElement("editeur").addText("editeur 1");
      livre = root.addElement("livre");
      livre.addElement("titre").addText("titre 2");
      livre.addElement("auteur").addText("auteur 2");
      livre.addElement("editeur").addText("editeur 2");
      livre = root.addElement("livre");
      livre.addElement("titre").addText("titre 3");
      livre.addElement("auteur").addText("auteur 3");
      livre.addElement("editeur").addText("editeur 3");
      OutputFormat format = OutputFormat.createPrettyPrint();
      XMLWriter writer = new XMLWriter( System.out, format );
      writer.write( document );
    } catch (Exception e){
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
C:\test_dom4j>java -cp .;c:\j2sdk1.4.0_01/jre/lib/ext/dom4j
-full.jar Testdom4j_6
<?xml version="1.0" encoding="UTF-8"?>
<bibliotheque>
  <livre>
    <titre>titre 1</titre>
    <auteur>auteur 1</auteur>
    <editeur>editeur 1</editeur>
  </livre>
  <livre>
    <titre>titre 2</titre>
    <auteur>auteur 2</auteur>
    <editeur>editeur 2</editeur>
  </livre>
  <livre>
    <titre>titre 3</titre>
    <auteur>auteur 3</auteur>
    <editeur>editeur 3</editeur>
  </livre>
</bibliotheque>
```

La classe OutputFormat possède une méthode createPrettyPrint() qui renvoie un objet de type OutputFormat contenant des paramètres par défaut.

Il est possible d'obtenir une chaîne de caractères à partir de tout ou partie d'un document

Exemple :

```
import org.dom4j.*;
import org.dom4j.io.*;
public class Testdom4j_7 {
  public static void main(String args[]) {
    Document document = DocumentHelper.createDocument();
    Element root = document.addElement( "bibliotheque" );
    Element livre = null;
    String texte = "";
    try {
      livre = root.addElement("livre");
      livre.addElement("titre").addText("titre 1");
      livre.addElement("auteur").addText("auteur 1");
      livre.addElement("editeur").addText("editeur 1");
      texte = document.asXML();
```

```
      System.out.println(texte);
    } catch (Exception e){
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
C:\test_dom4j>java -cp .;c:\j2sdk1.4.0_01/jre/lib/ext/dom4j
-full.jar Testdom4j_7
<?xml version="1.0" encoding="UTF-8"?>
<bibliotheque><livre><titre>titre 1</titre><auteur>auteur 1</auteur><editeur>edi
teur 1</editeur></livre></bibliotheque>
```

La suite de cette section sera développée dans une version future de ce document

37. JAXB (Java Architecture for XML Binding)

JAXB est une spécification qui permet de faire correspondre un document XML à un ensemble de classes et vice et versa via des opérations de sérialisation/désérialisation nommée marshaling/unmarshaling.

JAXB permet aux développeurs de manipuler un document XML sans à avoir connaître XML ou la façon dont un document XML est traitée comme cela est le cas avec SAX, DOM ou StAX. La manipulation du document XML se fait en utilisant des objets précédemment générés à partir d'une DTD pour JAXB 1.0 et d'un schéma XML du document à traiter pour JAXB 2.0.

Le page officiel de JAXB est à l'url : Java Architecture for XML Binding (JAXB)

37.1. JAXB 1.0

JAXB est l'acronyme de Java Architecture for XML Binding.

Le but de l'API et des spécifications JAXB est de faciliter la manipulation d'un document XML en générant un ensemble de classes qui fournissent un niveau d'abstraction plus élevé que l'utilisation de JAXP (SAX ou DOM). Avec ces deux API, toute la logique de traitements des données contenues dans le document est à écrire.

JAXB au contraire fournit un outil qui analyse un schéma XML et génère à partir de ce dernier un ensemble de classes qui vont encapsuler les traitements de manipulation du document.

Le grand avantage est de fournir au développeur un moyen de manipuler un document XML sans connaître XML ou les technologies d'analyse. Toutes les manipulations se font au travers d'objets java.

Ces classes sont utilisées pour faire correspondre le document XML dans des instances de ces classes et vice et versa : ces opérations se nomment respectivement unmarshalling et marshalling.

L'implémentation de référence de JAXB v1.0 est fournie avec le JSWDK 1.1.

Les exemples de ce chapitre utilisent cette implémentation de référence et le fichier XML suivant :

Exemple :

```
<bibliotheque>
  <livre>
    <titre>titre 1</titre>
    <auteur>auteur 1</auteur>
    <editeur>editeur 1</editeur>
  </livre>
  <livre>
    <titre>titre 2</titre>
    <auteur>auteur 2</auteur>
    <editeur>editeur 2</editeur>
  </livre>
  <livre>
    <titre>titre 3</titre>
    <auteur>auteur 3</auteur>
    <editeur>editeur 3</editeur>
```

```
  </livre>
</bibliotheque>
```

Le schéma XML correspondant à ce fichier XML est le suivant :

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
<xs:element name="bibliotheque">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="livre" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  </xs:element>
  <xs:element name="livre">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="titre"/>
        <xs:element ref="auteur"/>
        <xs:element ref="editeur"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="titre" type="xs:string"/>
  <xs:element name="auteur" type="xs:string"/>
  <xs:element name="editeur" type="xs:string"/>
</xs:schema>
```

37.1.1. La génération des classes

L'outil xjc permet d'analyser un schéma XML et de générer les interfaces et les classes qui vont permettre la manipulation d'un document XML qui respecte ce schéma. Cette opération se nomme binding.

La syntaxe de cet outil est très simple :

xjc [options] shema

schema est le nom d'un fichier contenant le schéma XML.

Les principales options sont les suivantes :

- -nv : ne pas réaliser une validation stricte du schéma fourni
- -d repertoire : permet de préciser le nom du répertoire qui va contenir les classes générées
- -p package : permet de préciser le nom du package qui va contenir les classes générées

Exemple :

```
C:\java\jaxb>xjc test.xsd
parsing a schema...
compiling a schema...
generated\impl\AuteurImpl.java
generated\impl\BibliothequeImpl.java
generated\impl\BibliothequeTypeImpl.java
generated\impl\EditeurImpl.java
generated\impl\LivreImpl.java
generated\impl\LivreTypeImpl.java
generated\impl\TitreImpl.java
generated\Auteur.java
generated\Bibliotheque.java
generated\BibliothequeType.java
generated\Editeur.java
```

```
generated\Livre.java
generated\LivreType.java
generated\ObjectFactory.java
generated\Titre.java
generated\bgm.ser
generated\jaxb.properties
```

L'exécution de la commande de l'exemple génère les fichiers suivants :

Generated
- Auteur.java
- bgm.ser
- Bibliotheque.java
- BibliothequeType.java
- Editeur.java
- jaxb.properties
- Livre.java
- LivreType.java
- ObjectFactory.java
- Titre.java
- generated\impl
  - AuteurImpl.java
  - BibliothequeImpl.java
  - BibliothequeTypeImpl.java
  - EditeurImpl.java
  - LivreImpl.java
  - LivreTypeImpl.java
  - TitreImpl.java

Sans précision, les fichiers générés le sont dans le répertoire "Generated".

Pour préciser un autre répertoire, il faut utiliser l'option -d :

xjc -d sources test.xsd

Les classes et interfaces sont générées dans le répertoire "sources/generated"

Si le répertoire précisé n'existe pas, une exception est levée.

Exemple :

```
C:\java\jaxb>xjc -d sources test.xsd
parsing a schema...
compiling a schema...
java.lang.reflect.InvocationTargetException
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.
java:39)
```

Pour éviter l'utilisation du répertoire "generated", il faut préciser un package pour les entités générées en utilisant l'option -p.

Exemple :

```
C:\java\jaxb>xjc -d sources -p com.jmdoudoux.test.jaxb test.xsd
parsing a schema...
compiling a schema...
com\moi\test\jaxb\Auteur.java
com\moi\test\jaxb\Bibliotheque.java
com\moi\test\jaxb\BibliothequeType.java
com\moi\test\jaxb\Editeur.java
```

```
com\moi\test\jaxb\Livre.java
com\moi\test\jaxb\LivreType.java
com\moi\test\jaxb\ObjectFactory.java
com\moi\test\jaxb\Titre.java
com\moi\test\jaxb\jaxb.properties
com\moi\test\jaxb\bgm.ser
com\moi\test\jaxb\impl\AuteurImpl.java
com\moi\test\jaxb\impl\BibliothequeImpl.java
com\moi\test\jaxb\impl\BibliothequeTypeImpl.java
com\moi\test\jaxb\impl\EditeurImpl.java
com\moi\test\jaxb\impl\LivreImpl.java
com\moi\test\jaxb\impl\LivreTypeImpl.java
com\moi\test\jaxb\impl\TitreImpl.java
```

Un objet de type factory et des interfaces pour chacun des éléments qui compose le document sont définis.

Pour chaque élément qui peut contenir d'autres éléments, des interfaces de XXXType sont créées (BibliothequeType et LivreType dans l'exemple).

L'interface BibliothequeType définit simplement un getter sur une collection qui contiendra tous les livres.

L'interface LivreType définit des getters et des setters sur les éléments auteur, editeur et titre.

Les interfaces Titre, Editeur et Auteur définissent un getter et un setter sur la valeur des données que ces éléments contiennent.

La classe ObjectFactory permet de créer des instances des différentes entités définies.

Le répertoire impl contient les classes qui implémentent ces interfaces. Ces classes sont spécifiques à l'implémentation des spécifications JAXB utilisées.

Pour pouvoir utiliser ces interfaces et ces classes, il faut les compiler en incluant tous les fichiers .jar contenus dans le répertoire lib de l'implémentation de JAXB au classpath.

37.1.2. L'API JAXB

L'API JAXB propose un framework composé de classes regroupées dans trois packages :

- javax.xml.bind : contient les interfaces principales et la classe JAXBContext
- javax.xml.bind.util : contient des utilitaires
- javax.xml.bind.helper : contient une implémentation partielle de certaines interfaces pour faciliter le développement d'une implémentation des spécifications de JAXB

37.1.3. L'utilisation des classes générées et de l'API

Pour pouvoir utiliser JAXP, il faut tout d'abord obtenir un objet de type JAXBContext qui est le point d'entrée pour utiliser l'API. Il faut utiliser la méthode newInstance() qui attend en paramètre le nom du package qui contient les interfaces générées (celui fournit au paramètre -p de la commande xjc).

Pour pouvoir créer en mémoire les objets qui représentent le document XML, il faut à partir de l'instance du type JAXBContext, appeler la méthode createUnmarshaller() qui renvoie un objet de type Unmarshaller.

L'appel de la méthode unmarshal() permet de créer les différents objets.

Pour parcourir le document, il suffit d'utiliser les différents objets instanciés.

Exemple :

```
package com.jmdoudoux.test.jaxb;
import javax.xml.bind.*;
import java.io.*;
import java.util.*;
public class TestJAXB {

  public static void main(String[] args) {

    try {
      JAXBContext jc = JAXBContext.newInstance("com.jmdoudoux.test.jaxb");
      Unmarshaller unmarshaller = jc.createUnmarshaller();
      unmarshaller.setValidating(true);

      Bibliotheque bibliotheque = (Bibliotheque) unmarshaller.unmarshal(new File("test.xml"));

      List livres = bibliotheque.getLivre();
      for (int i = 0; i < livres.size(); i++) {
        LivreType livre = (LivreType) livres.get(i);
        System.out.println("Livre ");
        System.out.println("Titre   : " + livre.getTitre());
        System.out.println("Auteur  : " + livre.getAuteur());
        System.out.println("Editeur : " + livre.getEditeur());
        System.out.println();
      }
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
Livre
Titre   : titre 1
Auteur  : auteur 1
Editeur : editeur 1
Livre
Titre   : titre 2
Auteur  : auteur 2
Editeur : editeur 2
Livre
Titre   : titre 3
Auteur  : auteur 3
Editeur : editeur 3
```

37.1.4. La création d'un nouveau document XML

Parmi les classes générées à partir du schéma XML, il y a la classe ObjectFactory qui permet de créer des instances des autres classes générées.

Exemple :

```
import javax.xml.bind.*;
import java.io.*;
import java.util.*;
public class TestJAXB2 {

  public static void main(String[] args) {
    try {
      ObjectFactory objFactory = new ObjectFactory();

      Bibliotheque bibliotheque = (Bibliotheque) objFactory.createBibliotheque();
      List livres = bibliotheque.getLivre();
      for (int i = 1; i < 4; i++) {
        LivreType livreType = objFactory.createLivreType();
        //    LivreType livre = objFactory.createLivreType();
        livreType.setAuteur("Auteur" + i);
```

```
        livreType.setEditeur("Editeur" + i);
        livreType.setTitre("Titre" + i);
        livres.add(livreType);
      }

    } catch (Exception e) {

    }
  }
}
```

37.1.5. La génération d'un document XML

Une fois la représentation en mémoire du document XML créée ou modifiée, il est fréquent de devoir l'envoyer dans un flux tel qu'un fichier pour conserver les modifications. Cette opération se nomme marshalling.

Pour réaliser cette opération, il faut tout d'abord obtenir un objet du type JAXBContext en utilisant la méthode newInstance(). Cette méthode demande en paramètre une chaîne de caractères indiquant le package des interfaces gérées à partir du schéma.

La méthode createMarshaller() permet d'obtenir un objet de type Marshaller. C'est cet objet qui va formater le document XML.

Il est possible de lui préciser des propriétés pour effectuer sa tache en utilisant la méthode setProperty(). Des constantes sont définies pour ces propriétés dont les principales sont :

- JAXB_ENCODING : permet de préciser le jeux de caractères d'encodage du document XML sous la forme d'une chaîne de caractères
- JAXB_FORMATTED_OUTPUT : booléen qui indique si le document XML doit être formaté

La valeur des propriétés doit être un objet.

L'appel de la méthode marshal() formate le document dont l'objet racine est fourni en premier paramètre. Il existe plusieurs surcharges de cette méthode pour préciser où est envoyé le résultat de la génération. Le second paramètre permet de préciser cette cible : un flux en sortie, un arbre DOM, des événements SAX.

Exemple :

```
package com.jmdoudoux.test.jaxb;
import javax.xml.bind.*;
import java.io.*;
import java.util.*;
public class TestJAXB3 {

  public static void main(String[] args) {
    try {

      ObjectFactory objFactory = new ObjectFactory();

      Bibliotheque bibliotheque = (Bibliotheque) objFactory.createBibliotheque();
      List livres = bibliotheque.getLivre();
      for (int i = 1; i < 4; i++) {
        LivreType livreType = objFactory.createLivreType();
        //    LivreType livre = objFactory.createLivreType();
        livreType.setAuteur("Auteur" + i);
        livreType.setEditeur("Editeur" + i);
        livreType.setTitre("Titre" + i);
        livres.add(livreType);
      }
      JAXBContext jaxbContext = JAXBContext.newInstance("com.jmdoudoux.test.jaxb");
      Marshaller marshaller = jaxbContext.createMarshaller();
      marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, new Boolean(true));
      Validator validator = jaxbContext.createValidator();
```

```
        marshaller.marshal(bibliotheque, System.out);
      } catch (Exception e) {
      }
    }
}
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<bibliotheque>
<livre>
<titre>Titre1</titre>
<auteur>Auteur1</auteur>
<editeur>Editeur1</editeur>
</livre>
<livre>
<titre>Titre2</titre>
<auteur>Auteur2</auteur>
<editeur>Editeur2</editeur>
</livre>
<livre>
<titre>Titre3</titre>
<auteur>Auteur3</auteur>
<editeur>Editeur3</editeur>
</livre>
</bibliotheque>
```

37.2. JAXB 2.0

JAXB 2.0 a été développé sous la JSR 222 et elle est incorporée dans Java EE 5 et dans Java SE 6.

Les fonctionnalités de JAXB 2.0 par rapport à JAXB 1.0 sont :

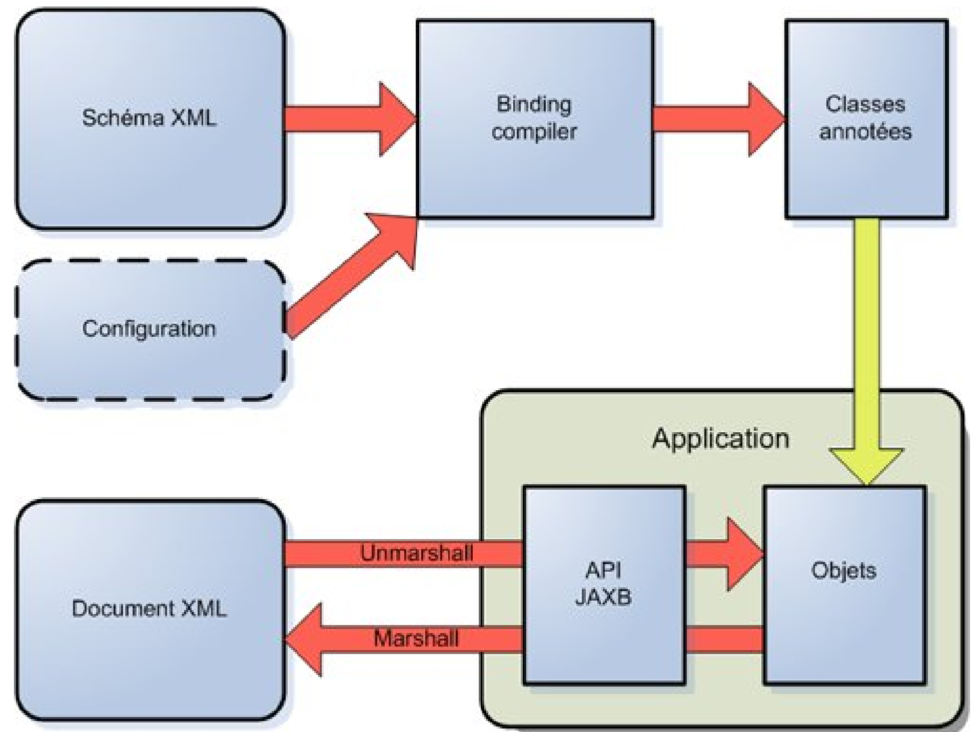
- support uniquement des schémas XML (les DTD ne sont plus supportées)
- mise en oeuvre des annotations
- assure la correspondance bidirectionnelle entre un schéma XML et le bean correspondant.
- l'utilisation de fonctionnalités proposées par Java 5 notamment les generics et les énumérations
- le nombre d'entités générées est moins important : JAXB 2.0 génère une classe pour chaque complexType du schéma alors que JAXB 1.0 génère une interface et une classe qui implémente cette interface. Une méthode de la classe ObjectFactory est générée pour renvoyer une instance de cette classe.

En plus de son utilité principale, JAXB 2.0 propose d'atteindre plusieurs objectifs :

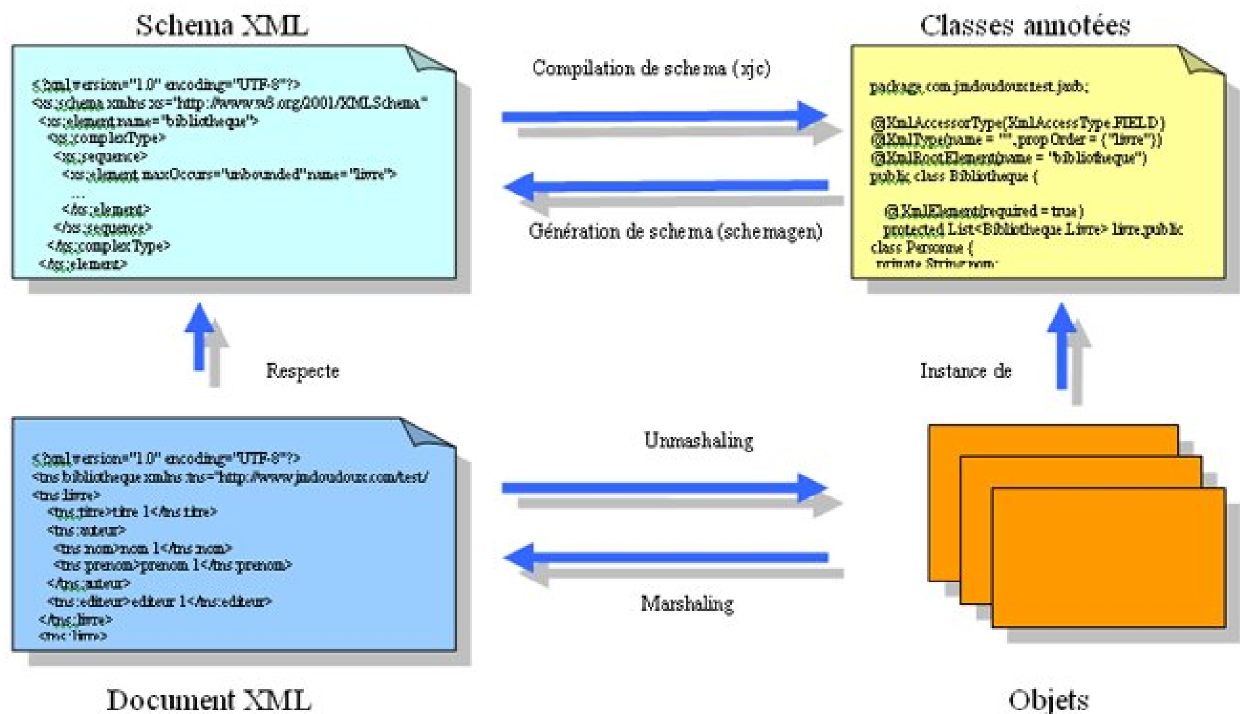
- Être facile à utiliser pour consulter et modifier un document XML sans connaissance ni de XML ni de techniques de traitement de documents XML
- Être configurable : JAXB met en oeuvre des fonctionnalités par défaut qu'il est possible de modifier par configuration pour répondre à ses propres besoins
- S'assurer que la création d'un document XML à partir d'objets et retransformer ce document en objets donne le même ensemble d'objets
- Pouvoir valider un document XML ou les objets qui encapsulent un document sans avoir à écrire le document correspondant
- Être portable : chaque implémentation doit au minimum mettre en oeuvre les spécifications de JAXB

L'utilisation de JAXB implique généralement deux étapes :

- Génération des classes et interfaces à partir du schéma XML
- Utilisation des classes générées et de l'API JAXB pour transformer un document XML en graphe d'objets et vice et versa, pour manipuler les données dans le graphe d'objets et pour valider le document



JAXB 2.0 permet toujours de mapper des objets Java dans un document XML et vice et versa. Il permet aussi de générer des classes Java à partir un schéma XML et vice et versa.



La sérialisation d'un graphe d'objets Java est effectué par une opération dite de mashalling. L'opération inverse est dite d'unmashalling. Lors de ces deux opérations, le document XML peut être validé.

JAXB 2.0 utilise de nombreuses annotations définies dans le package javax.xml.bin.annotation essentiellement pour préciser le mode de fonctionnement lors des opérations de marshaling/unmarshaling.

Ces annotations précisent le mapping entre les classes Java et le document XML. La plupart de ces annotations ont des valeurs par défaut ce qui réduit l'obligation de leur utilisation si la valeur par défaut correspond au besoin.

```
// This file was generated by the JavaTM
Architecture
// for XML Binding(JAXB) Reference
Implementation,
// vJAXB 2.0 in JDK 1.6
// See <a href="http://java.sun.com/xml/jaxb">
// http://java.sun.com/xml/jaxb</a>
// Any modifications to this file will be lost
// upon recompilation of the source schema.
// Generated on: 2007.06.20 at 02:16:59 PM CEST
package com.jmdoudoux.test.jaxb.perso;

import java.util.ArrayList;
import java.util.List;
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlType;


@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "", propOrder = {"livre"})
@XmlRootElement(name = "bibliotheque")
public class Bibliotheque {

  @XmlElement(required = true)
  protected List<Bibliotheque.Livre> livre;

  public List<Bibliotheque.Livre> getLivre() {
    if (livre == null) {
      livre = new
ArrayList<Bibliotheque.Livre>();
    }
    return this.livre;
  }

  @XmlAccessorType(XmlAccessType.FIELD)
  @XmlType(name = "",
    propOrder = {"titre","auteur","nomEditeur"})
  public static class Livre {

    @XmlElement(required = true)
    protected String titre;
    @XmlElement(required = true)
    protected Bibliotheque.Livre.Auteur auteur;
    @XmlElement(name = "editeur", required =
true)
    protected String nomEditeur;

    public String getTitre() {
      return titre;
    }

    public void setTitre(String value) {
      this.titre = value;
    }

    public Bibliotheque.Livre.Auteur getAuteur()
{
      return auteur;
    }

    public void setAuteur(
      Bibliotheque.Livre.Auteur value) {
      this.auteur = value;
    }

    public String getNomEditeur() {
      return nomEditeur;
    }

    public void setNomEditeur(String value) {
      this.nomEditeur = value;
    }

    @XmlAccessorType(XmlAccessType.FIELD)
    @XmlType(name = "",
      propOrder = {"nom","prenom"})
    public static class Auteur {

      @XmlElement(required = true)
      protected String nom;
      @XmlElement(required = true)
      protected String prenom;
```

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"

targetNamespace="http://www.jmdoudoux.com/test/jaxb/perso"
 xmlns:tns="http://www.jmdoudoux.com/test/jaxb"
 xmlns:xjc="http://java.sun.com/xml/ns/jaxb/xjc"
 jaxb:extensionBindingPrefixes="xjc"
 xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
jaxb:version="1.0"
 elementFormDefault="qualified">

 <xs:element name="bibliotheque">

  <xs:complexType>
   <xs:sequence>

    <xs:element maxOccurs="unbounded" name="livre">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="titre" type="xs:string" />


       <xs:element name="editeur" type="xs:string">
        <!--
         Personnalisation de la propriété
         editeur en nomEditeur
        -->
        <xs:annotation>
         <xs:appinfo>
          <jaxb:property name="nomEditeur" />
         </xs:appinfo>
        </xs:annotation>
       </xs:element>


       <xs:element name="auteur">
        <xs:complexType>
         <xs:sequence>
          <xs:element name="nom" type="xs:string" />
          <xs:element name="prenom" type="xs:string" />
         </xs:sequence>
        </xs:complexType>
       </xs:element>

      </xs:sequence>
     </xs:complexType>
    </xs:element>

   </xs:sequence>
  </xs:complexType>
 </xs:element>

</xs:schema>
```

JAXB 2.0 permet aussi de réaliser dynamiquement à l'exécution une transformation d'un graphe d'objets en document XML et vice et versa. C'est cette fonctionnalité qui est largement utilisée dans les services web via l'API JAX-WS 2.0.

La classe abstraite JAXBContext fournie par l'API JAXB permet de gérer la transformation d'objets Java en XML et vice et versa.

JAXB 2.0 propose plusieurs outils pour faciliter sa mise en oeuvre :

- un générateur de classes Java (schema compiler) à partir d'un schéma XML nommé xjc dans l'implémentation de référence. Ces classes générées mettent en oeuvre les annotations de JAXB.
- un générateur de schéma XML (schema generator) à partir d'un graphe d'objets nommé schemagen dans l'implémentation de référence.

L'API JAXB est contenue dans la package javax.xml.bind

37.2.1. L'obtention de JAXB 2.0

JAXB 2.0 est incorporée dans Java EE 5 et dans Java SE 6.

Pour les versions antérieures de ces plateformes, il est possible d'utiliser le Java Web Services Developer Pack 2.0 (JWSDP 2.0) qui contient l'implémentation de référence de JAXB 2.0.

Avec la version fournie avec JWSDP 2.0, les bibliothèques suivantes doivent être ajoutées au classpath :
jaxb\lib\jaxb-api.jar,
jaxb\lib\jaxb-impl.jar,
jaxb\lib\jaxb-xjc.jar,
jwsdp-shared\lib\activation.jar,
sjsxp\lib\jsr173_api.jar,
sjsxp\lib\sjsxp.jar

Attention JAXB 2.0 requiert un JDK 5.0 minimum pour être utilisé.

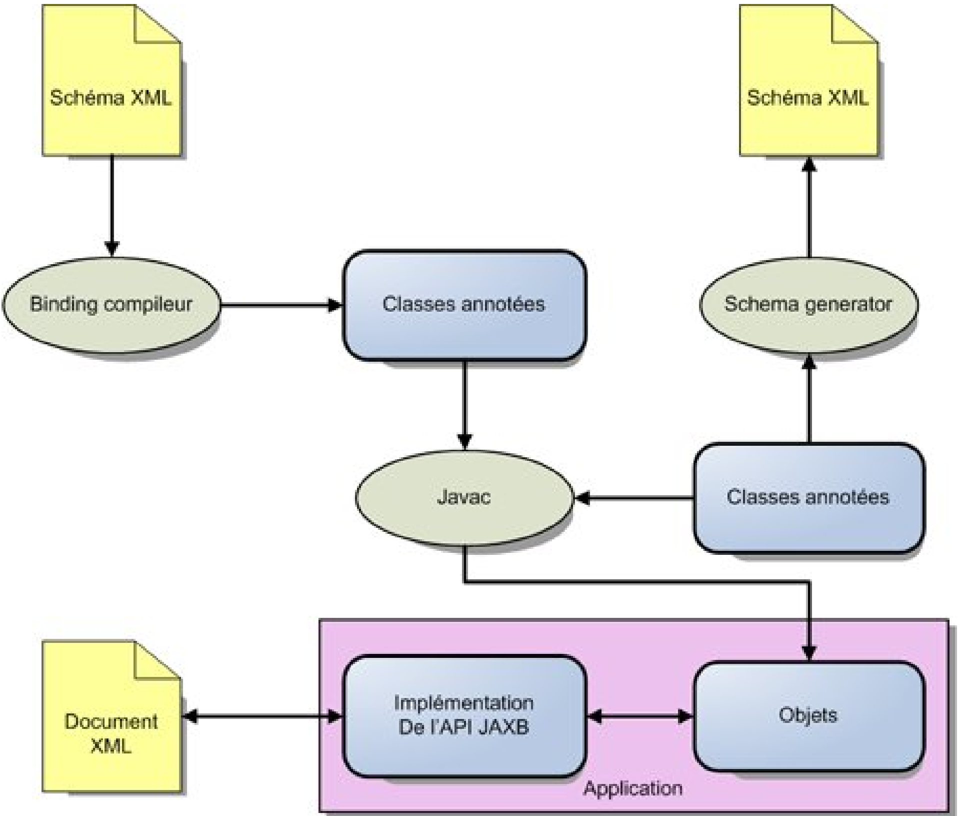
37.2.2. La mise en oeuvre de JAXB 2.0

La mise en oeuvre de JAXB requiert pour un usage standard plusieurs étapes :

- La génération des classes en utilisant l'outil xjc de JAXB à partir d'un schéma du document XML
- Ecriture de code utilisant les classes générées et l'API JAXB pour
- transformer un document XML en objets Java
- modifier des données encapsulées dans le graphe d'objets
- transformer le graphe d'objets en un document XML avec une validation optionnelle du document
- La compilation du code généré et écrit et l'exécution de l'application

L'utilisation de JAXB se fait donc en deux phases :

1. générer des classes à partir d'un schéma XML et utiliser ces classes dans le code de l'application
2. à l'exécution de l'application, le document XML est transformé en graphe d'objets, les données de ces objets peuvent être modifiées puis le document XML peut être regénéré à partir des objets.



L'utilisation de JAXB met en oeuvre plusieurs éléments :

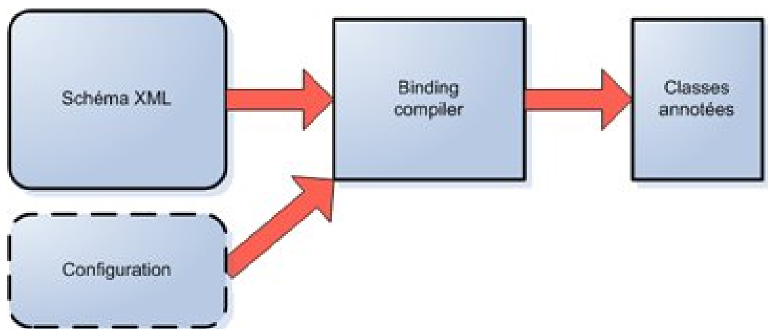
- Une ou plusieurs classes annotées qui vont encapsuler des données du document XML
- (optionnel) un schéma XML : c'est un document XML qui décrit la structure des éléments, attributs et entités d'un document XML. Le but d'un schéma XML est similaire à celui d'une DTD mais le schéma propose une description plus riche et plus fine. Ce schéma peut éventuellement être enrichi de données de configurations concernant les classes à générer
- (optionnel) un outil qui génère les classes annotées à partir d'un schéma avec éventuellement un fichier de configuration pour configurer les classes à générer
- Une API utilisée à l'exécution pour transformer un document XML en un ensemble d'objets du type des classes annotées et vice et versa et permettre des validations
- Un document XML qui sera lu et/ou écrit en fonction des traitements à réaliser

Les avantages d'utiliser JAXB sont nombreux :

- Facilite la manipulation de document XML dans une application Java
- Manipulation du document XML au travers d'objets : aucune connaissance de XML ou de la manière de traiter un document n'est requise
- Un document XML peut être créé en utilisant les classes générées
- Les traitements de JAXB peuvent être configurés
- Les ressources requises par le graphe d'objets utilisé par JAXB sont moins importantes qu'avec DOM

37.2.3. La génération des classes à partir d'un schéma

Pour permettre l'utilisation et la manipulation d'un document XML, JAXB propose de générer un ensemble de classes à partir du schéma XML du document.



Chaque implémentation de JAXB doit fournir un outil (binding compiler) qui permet la génération de classes et interfaces à partir d'un schema (binding a schema).

37.2.4. La commande xjc

L'implémentation de référence fournit l'outil xjc pour générer les classes à partir d'un schéma XML.

L'utilisation la plus simple de l'outil xjc est de lui fournir simplement le fichier qui contient le schéma XML du document à utiliser.

Exemple :

```
xjc biblio.xsd
```

L'outil xjc possède plusieurs options dont voici les principales :

Option	Rôle
-p nom_package	Précise le package qui va contenir les classes générées
-d répertoire	Précise le répertoire qui va contenir les classes générées
-nv	Inhibe la validation du schéma
-b fichier	Précise un fichier de configuration
-classpath chemin	Précise le classpath

37.2.5. Les classes générées

Le compilateur génère des classes en correspondance avec le schéma XML fourni à l'outil.

Exemple : biblio.xsd

```
<?xml version="1.0"  encoding="UTF-8"?>
<xs:schema  xmlns:xs="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://www.jmdoudoux.com/test/jaxb"
            xmlns:tns="http://www.jmdoudoux.com/test/jaxb"
            elementFormDefault="qualified">
  <xs:element  name="bibliotheque">
    <xs:complexType>
      <xs:sequence>
        <xs:element  maxOccurs="unbounded" name="livre">
          <xs:complexType>
            <xs:sequence>
              <xs:element  name="titre" type="xs:string" />
              <xs:element  name="auteur">
                <xs:complexType>
                  <xs:sequence>
```

```
                     <xs:element name="nom"  type="xs:string" />
                     <xs:element  name="prenom" type="xs:string" />
                   </xs:sequence>
                 </xs:complexType>
               </xs:element>
               <xs:element  name="editeur" type="xs:string" />
             </xs:sequence>
           </xs:complexType>
         </xs:element>
       </xs:sequence>
     </xs:complexType>
   </xs:element>
</xs:schema>
```

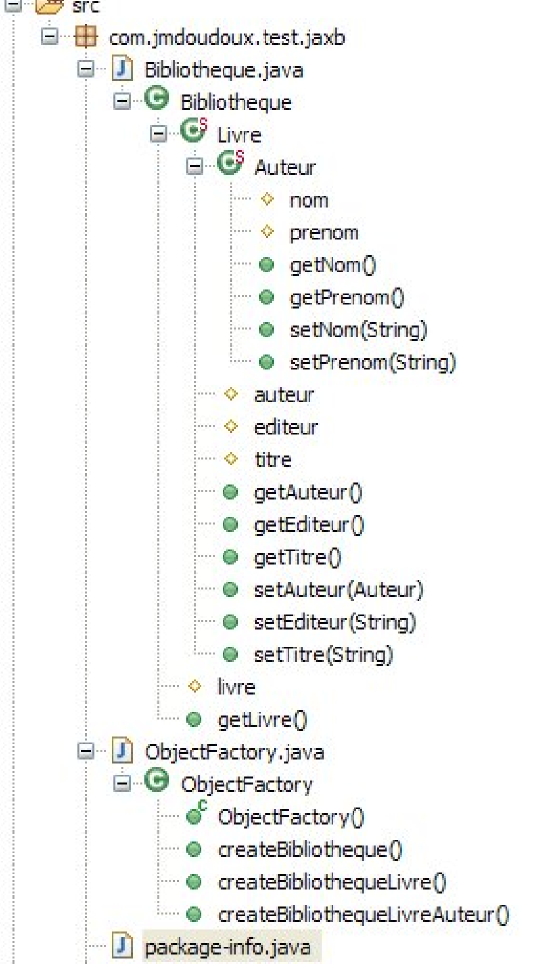
Exemple : exécution de la commande xjc

```
C:\Documents and Settings\jmd\workspace\TestJAXB>xjc -d src biblio.xsd
parsing a schema...
compiling a schema...
com\jmdoudoux\test\jaxb\Bibliotheque.java
com\jmdoudoux\test\jaxb\ObjectFactory.java
com\jmdoudoux\test\jaxb\package-info.java
```

Trois entités sont générées dans le répertoire src :

- com.jmdoudoux.test.jaxb.Bibliotheque.java : classes qui encapsulent le document XML
- com.jmdoudoux.test.jaxb.package-info.java : permet de conserver les espaces de nommage utilisés dans le package
- com.jmdoudoux.test.jaxb.ObjectFactory.java : fabrique qui permet d'instancier des objets utilisés lors du mapping

Par défaut, le package utilisé est déduit de l'espace de nommage défini dans le schéma.



Chaque classse qui encapsule un type complexe du schéma possède des getter et setter sur les éléments du schéma.

La fabrique permet de créer des instances de chacun des types d'objet correspondant à un type complexe du schéma. Cette fabrique est particulièrement utile lors de la création d'un nouveau document XML : le graphe d'objets est créé en ajoutant des instances des objets retournées par cette fabrique.

Les classes générées sont dépendantes de l'implémentation de JAXB utilisée : il est préférable d'utiliser les classes générées par une implémentation avec cette implémentation.

Par défaut, JAXB utilise des règles pour définir chaque entité incluse dans le schéma (element et complexType définis dans le schéma).

37.2.6. L'utilisation de l'API JAXB 2.0

JAXB fournit une API qui permet à l'exécution d'effectuer les opérations de transformation d'un document XML en un graphe d'objets utilisant les classes générées et vice et versa (unmashalling/marshalling) ainsi que des opérations de validation.

L'objet principal pour les opérations de transformation est l'objet JAXBContext : il permet d'utiliser l'API JAXB. Pour obtenir une instance de cet objet, il faut utiliser la méthode statique newInstance() en lui passant en paramètre le ou les packages contenant les classes générées à utiliser. Dans le cas où plusieurs packages doivent être précisés, il faut les séparer par une virgule.

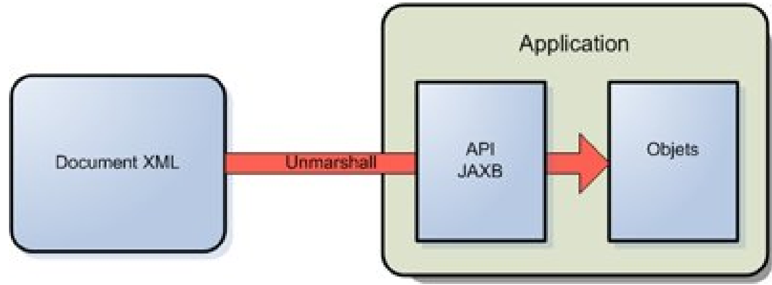
Une autre surcharge de la méthode newInstance() attend en paramètre la classe qui encapsule la racine du document.

Exemple :

```
JAXBContext jc = JAXBContext.newInstance("com.jmdoudoux.test.jaxb");
```

37.2.6.1. Le mapping d'un document XML à des objets (unmarshal)

L'API JAXB propose de transformer un document XML en un ensemble d'objets qui vont encapsuler les données et la hiérarchie du document. Ces objets sont des instances des classes générées à partir du schéma XML.



La création des objets nécessite la création d'un objet de type JAXBContext en utilisant la méthode statique newInstance().

Il faut ensuite instancier un objet de type Unmarshaller qui va permettre de transformer un document XML en un ensemble d'objets. Un telle instance est obtenue en utilisant la méthode createUnmarshaller() de la classe JAXBContext.

Exemple :

```
  Unmarshaller unmarshaller = jc.createUnmarshaller();
```

La méthode unmarshal() de la classe Unmarshaller se charge de traiter un document XML et retourne un objet du type complexe qui encapsule la racine du document XML. Elle possède de nombreuses surcharges à utiliser en fonction des

besoins.

Exemple :

```
  Bibliotheque bibliotheque = (Bibliotheque) unmarshaller.unmarshal(new File("biblio.xml"));
```

A partir de cet objet, il est possible d'obtenir et de modifier des données encapsulées dans les différents objets créés à partir des classes générées et du contenu du document. Chacun de ces objets possède des getter et des setter sur leur noeud direct.

Exemple :

```
List<Livre> livres = bibliotheque.getLivre();
      for (int i = 0; i < livres.size(); i++) {
      Livre livre = livres.get(i);
      System.out.println("Livre ");
      System.out.println("Titre : " + livre.getTitre());
      System.out.println("Auteur : " + livre.getAuteur().getNom()
        + " " +livre.getAuteur().getPrenom());
      System.out.println("Editeur : " + livre.getEditeur());
      System.out.println();
```

Il est possible de demander la validation du document en utilisant la méthode setValidating() de la classe Unmarshaller. Cela permet de demander la validation du document à traiter avec le schéma

Exemple :

```
      unmarshaller.setValidating(true);
```

Exemple : mise en oeuvre des entités générées à partir schéma biblio.xsd

```
package com.jmdoudoux.test.jaxb;

import java.io.File;
import java.util.List;

import javax.xml.bind.JAXBContext;
import javax.xml.bind.Unmarshaller;

import com.jmdoudoux.test.jaxb.Bibliotheque.Livre;

public class TestJAXB2 {

  public static void main(String[] args) {
    try {
      JAXBContext jc = JAXBContext.newInstance("com.jmdoudoux.test.jaxb");
      Unmarshaller unmarshaller = jc.createUnmarshaller();
      Bibliotheque bibliotheque = (Bibliotheque) unmarshaller.unmarshal(
             new File("biblio.xml"));
      List<Livre> livres = bibliotheque.getLivre();
      for (int i = 0; i < livres.size(); i++) {
      Livre livre = livres.get(i);
      System.out.println("Livre ");
      System.out.println("Titre : " + livre.getTitre());
      System.out.println("Auteur : " + livre.getAuteur().getNom()
        + " " +livre.getAuteur().getPrenom());
      System.out.println("Editeur : " + livre.getEditeur());
      System.out.println();
      }
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Exemple : le document XML utilisé

```
<?xml version="1.0"  encoding="UTF-8"?>
<tns:bibliotheque  xmlns:tns="http://www.jmdoudoux.com/test/jaxb"
                   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                   xsi:schemaLocation="http://www.jmdoudoux.com/test/jaxb biblio.xsd  ">
   <tns:livre>
     <tns:titre>titre 1</tns:titre>
     <tns:auteur>
       <tns:nom>nom 1</tns:nom>
       <tns:prenom>prenom 1</tns:prenom>
     </tns:auteur>
     <tns:editeur>editeur 1</tns:editeur>
   </tns:livre>
   <tns:livre>
     <tns:titre>titre 2</tns:titre>
     <tns:auteur>
       <tns:nom>nom 2</tns:nom>
       <tns:prenom>prenom 2</tns:prenom>
     </tns:auteur>
     <tns:editeur>editeur 2</tns:editeur>
   </tns:livre>
   <tns:livre>
     <tns:titre>titre 3</tns:titre>
     <tns:auteur>
       <tns:nom>nom 3</tns:nom>
       <tns:prenom>prenom 3</tns:prenom>
     </tns:auteur>
     <tns:editeur>editeur 3</tns:editeur>
   </tns:livre>
</tns:bibliotheque>
```

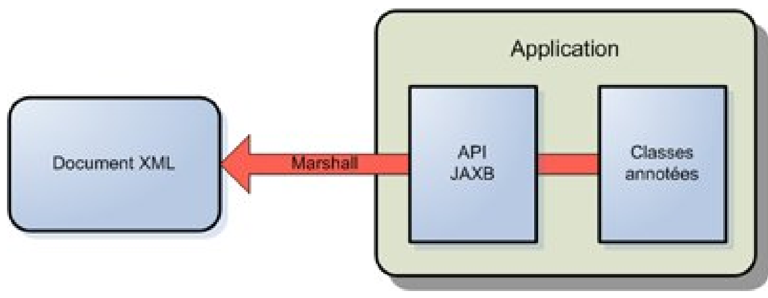
Résultat :

```
Livre
Titre : titre 1
Auteur : nom 1 prenom 1
Editeur : editeur 1

Livre
Titre : titre 2
Auteur : nom 2 prenom 2
Editeur : editeur 2

Livre
Titre : titre 3
Auteur : nom 3 prenom 3
Editeur : editeur 3
```

37.2.6.2. La création d'un document XML à partir d'objets

JAXB permet de créer un document XML à partir d'un graphe d'objets : cette opération est nommée marshalling. Une opération de mashalling est l'opération inverse de l'opération d'unmarshalling.

Ce graphe d'objets peut être issu d'une opération de type unmarshalling (construction à partir d'un document XML existant) ou issu d'une création de toutes pièces de l'ensemble des objets. Dans le premier cas cela correspond à une modification du document et dans le second cas à une création de document.



La création des objets nécessite la création d'un objet de type JAXBContext en utilisant la méthode statique newInstance().

Il faut ensuite instancier un objet de type Marshaller qui va permettre de transformer un document XML en un ensemble d'objets. Une telle instance est obtenue en utilisant la méthode createMarshaller() de la classe JAXBContext.

Exemple :

```
Marshaller marshaller = jc.createMarshaller();
```

La méthode marshal() de la classe marshaller se charge de créer un document XML à partir d'un graphe d'objets dont l'objet racine lui est fourni en paramètre.

La méthode marshal() possède plusieurs surcharges qui permettent de préciser la forme du document XML généré :

- un fichier (File),
- un flux (outputStream),
- un flux de caractères (Writer),
- un document DOM (Document),
- un gestionnaire d'événements SAX (ContentHandler),
- un objet de type javax.xml.transform.SAXResult,
- un objet de type javax.xml.transform.DOMResult,
- un objet de type javax.xml.transform.StreamResult,
- un objet de type javax.xml.stream.XMLStreamWriter,
- ou un objet de type javax.xml.stream.XMLEventWriter.

L'objet Mashaller possède des propriétés qu'il est possible de valoriser en utilisant la méthode setProperty(). Les spécifications de JAXB proposent des propriétés qui doivent être obligatoirement supportées par l'implémentation. Chaque implémentation est libre de proposer des propriétés supplémentaires.

Exemple : demander le formattage du document créé

```
Marshaller m = context.createMarshaller();
m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
```

Il est possible de demander la validation du graphe d'objets. La validation n'est pas intégrée à l'opération de mashalling mais elle est effectuée à la demande séparément

La validation s'effectue en utilisant la classe Validator. Une instance de cette classe est obtenue en utilisation la méthode createValidator() de la classe JAXBContext.

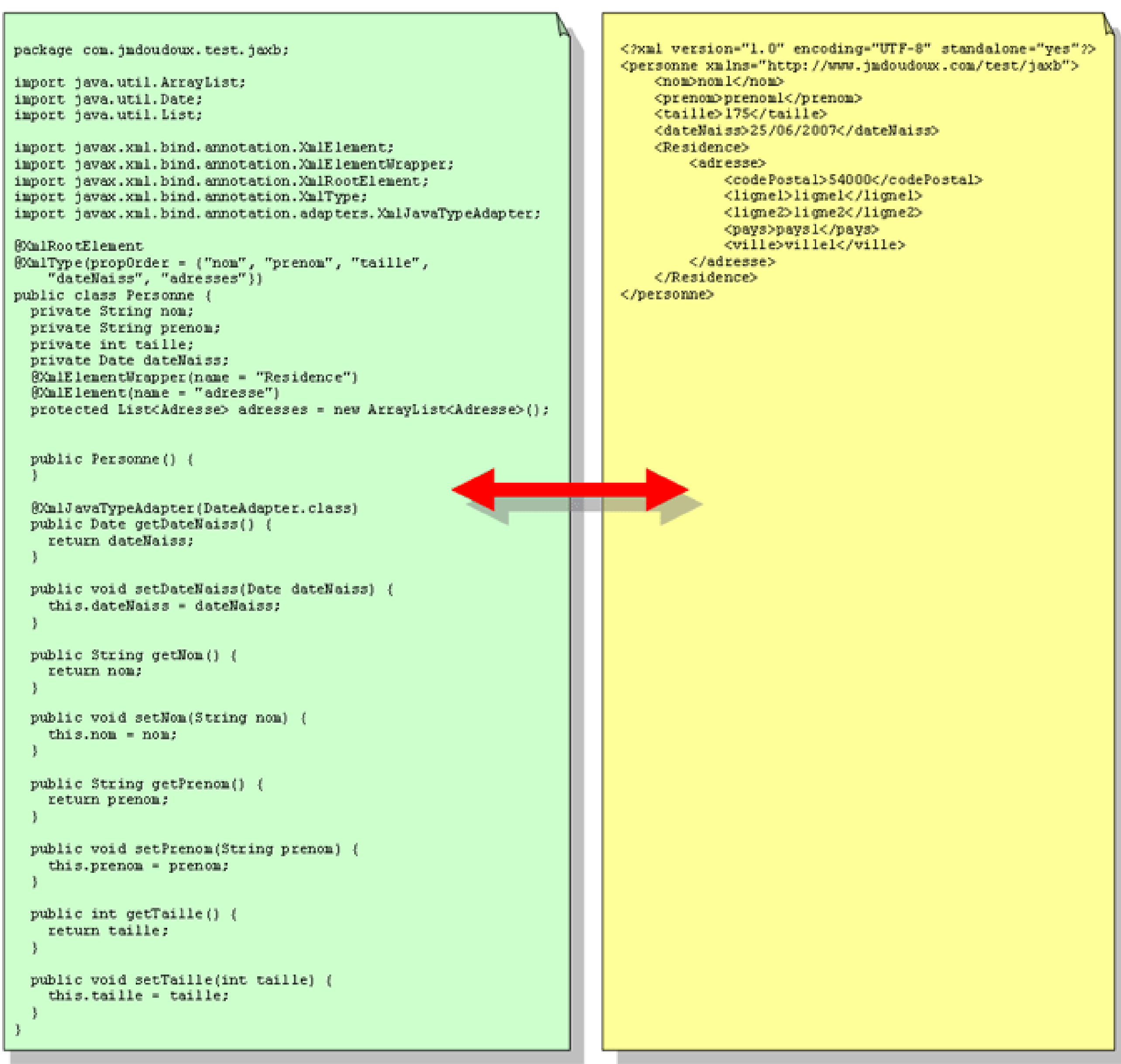
Exemple :

```
Validator validator = jaxbContext.createValidator();
```

Pour valider le graphe d'objets vis-à-vis du schéma, il faut utiliser la méthode validate() de la classe Validator en lui passant en paramètre l'objet qui encapsule la racine du document.

37.2.6.3. En utilisant des classes annotées

JAXB permet de mapper un document XML vers une ou plusieurs classes annotées sans être obligé d'utiliser un schéma XML. Dans ce cas, le développeur a la charge d'écrire la ou les classes annotées requises.



Il n'est donc pas nécessaire d'utiliser des classes générées par le compilateur de schéma mais dans ce cas, la ou les classes doivent être crées manuellement en utilisant les annotations adéquates.

La classe qui encapsule la racine du document doit être annotée avec l'annotation @XmlRootElement. Une exception de type javax.xml.bind.MarshalException est levée par JAXB si la classe racine ne possède pas cette annotation.

Exemple :

```
javax.xml.bind.MarshalException
 - with linked exception:
[com.sun.istack.internal.SAXException2: unable to marshal type "com.jmdoudoux.test.jaxb.
Personne" as an element because it is missing an @XmlRootElement annotation]
```

JAXB utilise des comportements par défaut qui ne nécessitent de la part du développeur que de définir des comportements particuliers si la valeur par défaut ne convient pas.

Exemple : un bean utilisé avec JAXB

```
package com.jmdoudoux.test.jaxb;

import java.util.Date;
import javax.xml.bind.annotation.XmlRootElement;
```

```
@XmlRootElement
public class Personne {
  private String nom;
  private String prenom;
  private int taille;
  private Date dateNaiss;

  public Personne() {
  }

  public Personne(String nom, String prenom, int taille, Date dateNaiss) {
    super();
    this.nom = nom;
    this.prenom = prenom;
    this.taille = taille;
    this.dateNaiss = dateNaiss;
  }

  public Date getDateNaiss() {
    return dateNaiss;
  }

  public void setDateNaiss(Date dateNaiss) {
    this.dateNaiss = dateNaiss;
  }

  public String getNom() {
    return nom;
  }

  public void setNom(String nom) {
    this.nom = nom;
  }

  public String getPrenom() {
    return prenom;
  }

  public void setPrenom(String prenom) {
    this.prenom = prenom;
  }

  public int getTaille() {
    return taille;
  }

  public void setTaille(int taille) {
    this.taille = taille;
  }
}
```

La création du document nécessite la création d'un objet de type JAXBContext en utilisant la méthode statique newInstance().

Il faut ensuite créer un objet de type Marshaller à partir du contexte et appeler sa méthode marshall pour générer le document.

Exemple : marshalling de l'objet

```
package com.jmdoudoux.test.jaxb;

import java.util.Date;

import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Marshaller;

public class TestJAXB1 {

  public static void main(String[] args) {
    try {
```

```
      JAXBContext context = JAXBContext.newInstance(Personne.class);
      Marshaller m = context.createMarshaller();
      m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
      Personne p = new Personne("nom1", "prenom1", 175, new Date());
      m.marshal(p, System.out);
    } catch (JAXBException ex) {
      ex.printStackTrace();
    }
  }
}
```

Résultat :

```
<?xml version="1.0"  encoding="UTF-8" standalone="yes"?>
<personne>
  <dateNaiss>2007-01-16T17:03:31.213+01:00</dateNaiss>
  <nom>nom1</nom>
  <prenom>prenom1</prenom>
  <taille>175</taille>
</personne>
```

37.2.6.4. En utilisant les classes générées à partir d'un schéma

Une des classes générées à partir du schéma se nomme ObjectFactory : c'est une fabrique d'objets pour les classes générées qui encapsulent des données d'un document respectant le schéma.

Pour créer un document XML en utilisant ces classes, il faut mettre en oeuvre plusieurs étapes.

La création du document nécessite la création d'un objet de type JAXBContext en utilisant la méthode statique newInstance().

Il faut ensuite créer le graphe d'objets en utilisant la classe ObjectFactory pour instancier les différents objets et valoriser les données de ces objets en utilisant les setter.

Il faut créer un objet de type Marshaller à partir du contexte et appeler sa méthode marshall pour générer le document.

Exemple :

```
package com.jmdoudoux.test.jaxb;

import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Marshaller;

public class TestJAXB3 {
  public static void main(String[] args) {
    try {
      JAXBContext context = JAXBContext.newInstance(Bibliotheque.class);
      Marshaller m = context.createMarshaller();
      m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);

      ObjectFactory fabrique = new ObjectFactory();
      Bibliotheque bibliotheque = fabrique.createBibliotheque();

      Bibliotheque.Livre livre = fabrique.createBibliothequeLivre();
      livre.setEditeur("editeur 1");
      livre.setTitre("titre 1");

      Bibliotheque.Livre.Auteur auteur = fabrique
          .createBibliothequeLivreAuteur();
      auteur.setNom("nom 1");
      auteur.setPrenom("prenom 1");
      livre.setAuteur(auteur);

      bibliotheque.getLivre().add(livre);
```

```
            m.marshal(bibliotheque, System.out);
        } catch (JAXBException ex) {
            ex.printStackTrace();
        }
    }
}
```

37.2.7. La configuration de la liaison XML / Objets

JAXB propose des fonctionnalités pour configurer finement les traitements qu'il propose.

37.2.7.1. l'annotation du schéma XML

JAXB utilise des traitements par défaut qu'il est possible de configurer différemment en utilisant soit les annotations soit un fichier de configuration qui sera fourni au compilateur.

Les classes générées peuvent aussi être configurées, notamment le nom du package et des classes utilisées.

Par défaut, le générateur de classes à partir du schéma utilise des conventions de nommage des différentes entités générées à partir des noms utilisés dans le schéma. Il est possible de configurer de façon différente les noms utilisés.

Les spécifications JAXB décrivent de façon précise comment les éléments d'un schéma sont transformés en classes Java. Il est possible de préciser des informations particulières dans le schéma pour modifier ce comportement par défaut.

Ces informations peuvent être incluse directement dans le schéma ou fournie dans un fichier dédié. Dans le schéma, ces informations sont fournies dans un tag <annotation> qui contient un tag fils <appinfo>.

Exemple : biblio2.xsd

```
<?xml  version="1.0" encoding="UTF-8"?>
<xs:schema  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.jmdoudoux.com/test/jaxb/perso"
   xmlns:tns="http://www.jmdoudoux.com/test/jaxb"
   xmlns:xjc="http://java.sun.com/xml/ns/jaxb/xjc"
  jaxb:extensionBindingPrefixes="xjc"
   xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"  jaxb:version="1.0"
  elementFormDefault="qualified">
  <xs:element  name="bibliotheque">
    <xs:complexType>
      <xs:sequence>
        <xs:element  maxOccurs="unbounded" name="livre">
          <xs:complexType>
            <xs:sequence>
              <xs:element  name="titre" type="xs:string" />
               <xs:element  name="auteur">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element  name="nom"
                       type="xs:string" />
                    <xs:element  name="prenom"
                      type="xs:string" />
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
              <xs:element  name="editeur" type="xs:string">
                <!--
                  Personnalisation de la  propriété editeur en nomEditeur
                -->
                <xs:annotation>
                  <xs:appinfo>
                    <jaxb:property  name="nomEditeur" />
                  </xs:appinfo>
```

```
                </xs:annotation>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Exemple : génération des classes à partir du schéma annoté

```
C:\Documents and Settings\jmd\workspace\TestJAXB>xjc -d src -extension biblio2.x
sd
parsing a schema...
compiling a schema...
com\jmdoudoux\test\jaxb\perso\Bibliotheque.java
com\jmdoudoux\test\jaxb\perso\ObjectFactory.java
com\jmdoudoux\test\jaxb\perso\package-info.java
```

L'option -extension du générateur de classes autorise l'outil à utiliser des extensions proposées par l'implémentation de JAXB utilisée. Sans cette option, l'outil utilise le mode strict et une exception est levée si une extension est utilisée.

Exemple : la classe Livre générée à partir du schéma

```
...
    @XmlAccessorType(XmlAccessType.FIELD)
    @XmlType(name = "", propOrder = {
        "titre",
        "auteur",
        "nomEditeur"
    })
    public static class Livre {

        @XmlElement(required = true)
        protected String titre;
        @XmlElement(required = true)
        protected Bibliotheque.Livre.Auteur auteur;
        @XmlElement(name = "editeur", required = true)
        protected String nomEditeur;

...

        /**
         * Gets the value of the nomEditeur property.
         *
         * @return
         *     possible object is
         *     {@link String }
         *
         */
        public String getNomEditeur() {
            return nomEditeur;
        }

        /**
         * Sets the value of the nomEditeur property.
         *
         * @param value
         *     allowed object is
         *     {@link String }
         *
         */
        public void setNomEditeur(String value) {
            this.nomEditeur = value;
        }
...
```

JAXB propose de nombreuses fonctionnalités de configuration : consultez les spécifications pour de plus amples détails.

37.2.7.2. L'annotation des classes

La configuration de la transformation d'un document XML en objets Java est réalisée grâce à l'utilisation d'annotations dédiées dans les classes Java.

De nombreuses annotations sont définies par JAXB 2.0 dont voici les principales :

XmlAccessorOrder	Ordonner les champs et propriétés dans la classe
XmlAccessorType	Préciser comment un champ ou une propriété est sérialisé
XmlAnyAttribute	
XmlAnyElement	
XmlAttachmentRef	
XmlAttribute	Convertir une propriété en un attribut dans le document XML
XmlElement	Convertir une propriété en un élément dans le document XML
XmlElementDecl	Associer une fabrique à une élément XML
XmlElementRef	
XmlElementRefs	
XmlElements	
XmlElementWrapper	Créer un élément père dans le document XML pour des collections d'éléments
XmlEnum	
XmlEnumValue	
XmlID	
XmlIDREF	
XmlInlineBinaryData	
XmlList	
XmlMimeType	
XmlMixed	
XmlNs	Associer un prefixe d'un espace de nommage à un URI
XmlRegistry	Marquer une classe comme possédant une ou des méthodes annotées avec @XmlElementDecl
XmlRootElement	Associer une classe ou une énumération à un élément XML
XmlSchema	Associer un espace de nommage à un package
XmlSchemaType	Associer un type Java ou une énumeration à un type défini dans un schéma
XmlSchemaTypes	
XmlTransient	Marquer une entité pour ne pas être mappée dans le document XML
XmlType	
XmlValue	

Ces annotations sont définies dans le package javax.xml.bind.annotation.

L'annotation @XmlRootElement peut être utilisée sur une classe pour préciser que cette classe sera le tag racine du document XML. Chaque attribut de la classe sera un tag fils dans le document XML. L'attribut namespace de l'annotation @XmlRootElement permet de préciser l'espace de nommage.

L'annotation XmlTransient permet d'ignorer une entité dans le mapping.

Exemple :

```
  @XmlTransient
  public String getNom() {
    return nom;
  }
```

Résultat :

```
<?xml version="1.0" encoding="UTF-8"  standalone="yes"?>
<personne  xmlns="http://www.jmdoudoux.com/test/jaxb">
     <dateNaiss>2007-06-21T15:18:28.809+02:00</dateNaiss>
    <prenom>prenom1</prenom>
    <taille>175</taille>
</personne>
```

L'annotation XmlAttribute permet de mapper une propriété sous la forme d'un attribut et fournir des précisions sur la configuration de cet attribut

Exemple :

```
@XmlAttribute (name="nomPrincipal")
public String getNom() {
  return nom;
```

Résultat :

```
<?xml version="1.0"  encoding="UTF-8" standalone="yes"?>
<personne  xmlns="http://www.jmdoudoux.com/test/jaxb" nom="nom1">
  <dateNaiss>2007-06-21T15:20:34.377+02:00</dateNaiss>
  <prenom>prenom1</prenom>
  <taille>175</taille>
</personne>
```

Pour les collections, il est possible d'utiliser l'annotation @XmlElementWrapper pour définir un élement père qui encapsule les occurrences de la collection et d'utiliser l'annotation XmlElement pour préciser le nom de chaque élément de la collection

Exemple :

```
  @XmlElementWrapper(name = "Residence")
  @XmlElement(name = "adresse")
  protected List<Adresse> adresses = new ArrayList<Adresse>();
```

Résultat :

```
<?xml version="1.0"  encoding="UTF-8" standalone="yes"?>
<personne  xmlns="http://www.jmdoudoux.com/test/jaxb">
  <Residence>
    <adresse>
      <codePostal>54000</codePostal>
      <ligne1>ligne1</ligne1>
      <ligne2>ligne2</ligne2>
      <pays>pays1</pays>
      <ville>ville1</ville>
    </adresse>
  </Residence>
  <dateNaiss>2007-06-21T15:34:35.547+02:00</dateNaiss>
  <nom>nom1</nom>
  <prenom>prenom1</prenom>
  <taille>175</taille>
</personne>
```

Il est possible de configurer l'ordre des éléments

Exemple :

```
@XmlRootElement
@XmlType(propOrder = {"nom", "prenom", "taille",
"dateNaiss", "adresses"})
public class Personne {
  private String nom;
  private String prenom;
  private int taille;
  private Date dateNaiss;
  @XmlElementWrapper(name = "Residence")
  @XmlElement(name = "adresse")
  protected List<Adresse> adresses = new ArrayList<Adresse>();
```

Résultat :

```
<?xml version="1.0"  encoding="UTF-8" standalone="yes"?>
<personne  xmlns="http://www.jmdoudoux.com/test/jaxb">
  <nom>nom1</nom>
  <prenom>prenom1</prenom>
  <taille>175</taille>
   <dateNaiss>2007-06-21T15:44:12.815+02:00</dateNaiss>
  <Residence>
     <adresse>
        <codePostal>54000</codePostal>
        <ligne1>ligne1</ligne1>
        <ligne2>ligne2</ligne2>
        <pays>pays1</pays>
        <ville>ville1</ville>
     </adresse>
  </Residence>
</personne>
```

Il est possible de définir des classes de type Adapter qui permettent de personnaliser la façon dont un objet est serialisé/desérialisé dans le document XML.

Ces adapters héritent de la classe javax.xml.bind.annotation.adapters.XmlAdapter. Il suffit de redéfinir les méthodes marshal() et unmashal(). Ces méthodes seront utilisées à l'exécution par l'API JAXB lors des transformations.

Exemple : la classe DateAdapter

```
package com.jmdoudoux.test.jaxb;

import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Date;

import javax.xml.bind.annotation.adapters.XmlAdapter;

public class DateAdapter extends XmlAdapter<String, Date> {

  DateFormat df = new SimpleDateFormat("dd/MM/yyyy");

  public Date unmarshal(String date) throws Exception {
    return df.parse(date);
  }

  public String marshal(Date date) throws Exception {
    return df.format(date);
  }
}
```

L'annotation @XmlJavaTypeAdapter permet de préciser la classe de type Adapter qui sera à utiliser plutôt que d'utiliser la conversion par défaut.

Exemple :

```
package com.jmdoudoux.test.jaxb;

import java.util.ArrayList;
import java.util.Date;
import java.util.List;

import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlElementWrapper;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlType;
import javax.xml.bind.annotation.adapters.XmlJavaTypeAdapter;

@XmlRootElement
@XmlType(propOrder = {"nom", "prenom", "taille",
    "dateNaiss", "adresses"})
public class Personne {
  private String nom;
  private String prenom;
  private int taille;
  private Date dateNaiss;
  @XmlElementWrapper(name = "Residence")
  @XmlElement(name = "adresse")
  protected List<Adresse> adresses = new ArrayList<Adresse>();


  public Personne() {
  }

  public Personne(String nom, String prenom, int taille, Date dateNaiss) {
    super();
    this.nom = nom;
    this.prenom = prenom;
    this.taille = taille;
    this.dateNaiss = dateNaiss;

  }

  @XmlJavaTypeAdapter(DateAdapter.class)
  public Date getDateNaiss() {
    return dateNaiss;
  }

  public void setDateNaiss(Date dateNaiss) {
    this.dateNaiss = dateNaiss;
  }
```

Résultat :

```
<?xml version="1.0"  encoding="UTF-8" standalone="yes"?>
<personne  xmlns="http://www.jmdoudoux.com/test/jaxb">
  <nom>nom1</nom>
  <prenom>prenom1</prenom>
  <taille>175</taille>
  <dateNaiss>22/06/2007</dateNaiss>
  <Residence>
    <adresse>
       <codePostal>54000</codePostal>
      <ligne1>ligne1</ligne1>
      <ligne2>ligne2</ligne2>
      <pays>pays1</pays>
      <ville>ville1</ville>
    </adresse>
  </Residence>
</personne>
```

Ceci ne présente qu'une toute petite partie des fonctionnalités proposées par JAXB. Pour de plus amples informations, consultez les spécifications de JAXB.

37.2.7.3. La génération d'un schéma à partir de classes compilées

L'outil schemagen fourni avec l'implémentation par défaut permet de générer un schéma XML à partir de classes annotées compilées

Exemple :

```
C:\Documents and Settings\jmd\workspace\TestJAXB>schemagen -cp ./bin com.jmdoudo
ux.test.jaxb.Personne
Note: Writing schema1.xsd
```

Résultat :

```
<?xml  version="1.0" encoding="UTF-8"  standalone="yes"?>
<xs:schema  version="1.0"  xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="personne" type="personne"/>
  <xs:complexType  name="personne">
      <xs:sequence>
        <xs:element name="nom"  type="xs:string" minOccurs="0"/>
        <xs:element name="prenom"  type="xs:string" minOccurs="0"/>
        <xs:element name="taille"  type="xs:int"/>
        <xs:element name="dateNaiss"  type="xs:string" minOccurs="0"/>
        <xs:element name="Residence"  minOccurs="0">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="adresse" type="adresse" maxOccurs="unbounded" minOccurs="0"/>
            </xs:sequence>
           </xs:complexType>
         </xs:element>
       </xs:sequence>
     </xs:complexType>
     <xs:complexType name="adresse">
       <xs:sequence>
         <xs:element name="codePostal" type="xs:string"  minOccurs="0"/>
         <xs:element name="ligne1" type="xs:string"  minOccurs="0"/>
         <xs:element name="ligne2" type="xs:string"  minOccurs="0"/>
         <xs:element name="pays" type="xs:string"  minOccurs="0"/>
         <xs:element name="ville" type="xs:string"  minOccurs="0"/>
       </xs:sequence>
     </xs:complexType>
</xs:schema>
```

38. StAX (Streaming Api for XML)

StAX est l'acronyme de Streaming Api for XML : c'est une API qui permet de traiter un document XML de façon simple en consommant peu de mémoire tout en permettant de garder le contrôle sur les opérations d'analyse ou d'écriture.

StAX a été développée sous la JSR-173 et est incorporée dans Java SE 6.0.

StAX propose des fonctionnalités pour parcourir et écrire un document XML mais ne permet pas de manipuler le contenu d'un document.

Le but de StAX n'est pas de remplacer SAX ou DOM mais de proposer une nouvelle façon d'analyser un document XML : StAX vient en complément des API DOM et SAX.

Sa mise en oeuvre par rapport aux deux API existantes peut être dans certains cas plus simple et donc plus facile que SAX et plus efficace et performante que DOM. StAX permet de traiter un document XML de manière rapide, facile et consommant peu de ressources : le modèle d'événements utilisé est plus simple que celui de SAX et les ressources requises sont moins importantes que pour un traitement via DOM.

38.1. La présentation de StAX

Avant StAX, l'analyse d'un document XML pouvait se faire principalement via deux API standards (DOM et SAX) ou une API non standard (JDOM).

L'analyse d'un document XML peut être classée en deux grandes catégories de parseur :

- Parseur basé sur un arbre : DOM implémente cette technique qui consiste à représenter et stocker le document dans un arbre d'objets. Il est ainsi possible, une fois cet arbre créé, de parcourir librement les noeuds de l'arbre et de le modifier. Cette représentation est généralement plus gourmande en ressource que le document lui-même ce qui la rend particulièrement inadapté à des documents de grande taille.
- Parseur basé sur un flux (document streaming) : des événements sont émis lors de la lecture séquentielle du flux pour notifier chaque changement lors de l'analyse du document. SAX implémente cette technique qui nécessite moins de ressources mais offre cependant moins de souplesse dans la manipulation du document

Il existe deux sortes de traitement par flux :

- Push : le parser émet des événements au client à chaque noeud du document rencontré que le client en ait besoin ou non
- Pull : le client demande explicitement au parser de lui donner l'événement suivant ce qui permet au client de conserver la main sur les traitements du parser et ainsi de piloter l'analyse

	Push parsing	Pull parsing
Implémentation	SAX	StAX
Contrôle des traitements	Par le parseur	Par le client

Complexité de mise en oeuvre	Moyenne	Faible
Parcours de tout le document	Oui	Non (interruption possible par le client)

Avec le traitement par flux, seul l'élément courant durant le parcours séquentiel du document est accessible. Ceci limite les traitements possibles sur le document et impose généralement de conserver un contexte.

L'utilisation d'un traitement par flux est particulièrement utile lors de la manipulation de gros documents, de l'utilisation dans un environnement possédant des ressources limitées (exemple utilisation avec Java ME) ou lors de traitements en parallèle de documents (exemple dans un serveur d'applications ou un moteur de services web).

StAX propose un modèle de traitement du document qui repose sur une lecture séquentielle du document sous le contrôle de l'application (ce n'est pas le parseur qui pilote le parcours mais l'application qui pilote le parseur). StAX représente un document sous la forme d'un ensemble d'événements qui sont fournis à la demande de l'application dans l'ordre du parcours séquentiel du document.

StAX repose sur le modèle de conception Iterator : chaque élément du document est parcouru séquentiellement à la demande du code pour émettre un événement. Ce parcours se fait à l'aide d'un curseur virtuel.

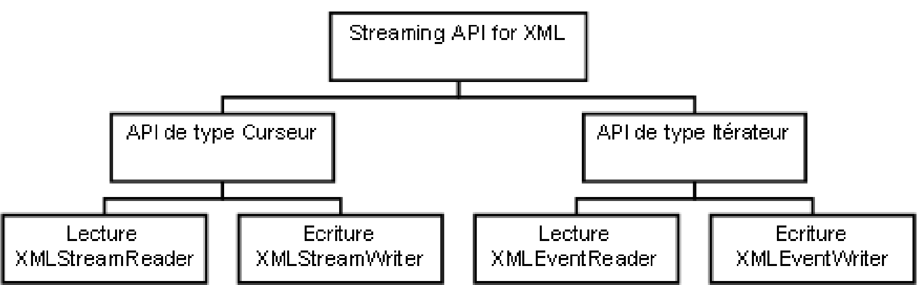
38.2. Les deux API de StAX

StAX est donc une API qui propose de mettre en oeuvre une troisième méthode pour traiter un document XML : le pull parsing. Son but est de fournir un parser qui puisse traiter de gros documents XML, avec une faible quantité de ressources requises et que ce soit le code qui pilote les traitements d'analyse et non le parser.

StAX propose une API pour un traitement d'un document XML sous la forme d'une itération sur des événements émis par le parser à la demande du client. Elle propose deux formes d'API :

- une API du type curseur (Cursor) : permet le parcours de chaque événement émis lors du parcours du document sous la forme d'entier
- une API de type itérateur (Event Iterator) : permet le parcours de chaque événement émis lors du parcours du document sous la forme d'un objet de type XMLEvent

Elles permettent toutes les deux la lecture et l'écriture d'un document XML.



La définition de deux API permet de les conserver séparément avec une faible complexité plutôt que d'avoir une seule API plus complexe.

L'API de type curseur parcours le flux du document et émet des événements sous la forme d'un entier dédié à chaque événement.

L'API de type curseur est plus efficace dans la mesure où elle n'a pas besoin d'instancier un objet pour chaque événement comme le fait l'API de type itérateur. L'API de type itérateur est plus facile à utiliser puisque toutes les données utiles sont déjà présent dans l'objet de type XMLEvent.

L'interface XMLStreamReader définit un contrat pour un objet qui va analyser un document XML avec une API de type curseur.

L'interface XMLStreamReader propose des méthodes pour obtenir des informations sur l'élément courant représenté par l'événement courant du curseur. Ces méthodes retournent des chaînes de caractères ce qui limite les ressources à une transformation en chaîne de caractères si les objets retournés étaient d'un autre type.

L'interface XMLStreamWriter définit un contrat pour un objet qui va générer un document XML.

L'API de type itérateur parcours le flux du document et émet des événements sous la forme d'objets de type XMLEvent qui encapsulent les informations de l'événement.

L'interface XMLEventReader définit un contrat pour un objet qui va analyser un document XML avec une API de type itérateur sur des événements. Elle hérite de l'interface Iterator : elle propose donc la méthode nextEvent() qui retourne le prochain événement et la méthode hasNext() qui permet de savoir si il y a encore un événement à traiter.

L'interface XMLEvent encapsule les données d'un événement lié au parcours du document XML : ces événements sont émis à la demande du client dans l'ordre de leur apparition lors du parcours du document.

La définition de deux API permet de laisser au développeur le choix d'utiliser l'API de type curseur pour limiter l'instanciation d'objets durant l'analyse du document ou d'utiliser l'API de type itérateur pour bénéficier directement des événements sous la forme d'objets. Le développeur peut ainsi choisir en fonction de son contexte de mettre en oeuvre l'une ou l'autre des API selon des critères de consommation de ressources ou de simplicité et de fiabilité du code.

L'API de type curseur est moins verbeuse et moins puissante que celle de type itérateur d'événements. Elle est cependant plus efficace car elle instancie moins d'objets. Le code à produire avec l'API de type curseur est plus petit et généralement plus efficace. L'API de type itérateur est plus flexible et évolutive que l'API de type curseur.

Elles permettent toutes les deux uniquement la lecture vers l'avant du document mais l'API de type itérateur propose en plus la méthode peek() qui permet de connaître le prochain événement.

StAX permet aussi de construire un document XML en utilisant les flux. Les API de type curseur et itérateur proposent leur propre interface pour permettre l'écriture de document.

Les API de StAX sont contenues dans les packages javax.xml.stream et javax.xml.transform.stream

Comme SAX, StAX est un parseur dont les spécifications sont écrites pour Java. Il existe une implémentation de référence mais l'implémentation de ces spécifications peut être réalisée par un tiers.

38.3. Les fabriques

StAX propose des fabriques pour les différents types d'objets : XMLInputFactory, XMLOutputFactory et XMLEventFactory. Des paramètres propres à une implémentation peuvent être manipulés en utilisant les méthodes getProperty() et setProperty() de ces fabriques.

La classe XMLInputFactory est une fabrique qui permet d'obtenir et de configurer une instance du parseur pour une lecture d'un document.

Il faut utiliser la méthode statique newInstance() pour obtenir une instance de la fabrique : elle détermine la classe à instancier en regardant dans l'ordre :

- Utilisation de la propriété système javax.xml.stream.XMLInputFactory
- Utilisation du fichier lib/xml.stream.properties dans le répertoire d'installation du JRE
- Utilisation de l'API Services avec le fichier META-INF/services/javax.stream.XMLInputFactory du jar
- Utilisation de l'instance par défaut

L'instance de la classe XMLInputFactory permet de configurer et d'instancier un parseur. La configuration se fait en utilisant des propriétés de la fabrique :

Propriété	Rôle
javax.xml.stream.isValidating	

	Permettre d'activer la validation du document en utilisant sa DTD (optionnelle , false par défaut)
javax.xml.stream.isCoalescing	Permettre d'indiquer si tous les événements de type characters contigus soient regroupés en un seul événement (false par défaut)
javax.xml.stream.isNamespaceAware	Supprimer le support des espaces de nommages (optionnelle, true par défaut)
javax.xml.stream.isReplacingEntityReferences	Permettre de demander le remplacement des entités de référence internes par leur valeur et ainsi émettre un événement de type de Characters (true par défaut)
javax.xml.stream.isSupportingExternalEntities	
javax.xml.stream.reporter	
javax.xml.stream.resolver	Permettre de préciser une implémentation de type XMLResolver utilisée pour résoudre les entités externes
javax.xml.stream.allocator	
javax.xml.stream.supportDTD	Permettre de préciser si le support des DTD est activé (true par défaut)

La classe XMLOutputFactory est une fabrique qui permet de créer des objets pour écrire un document.

Il faut utiliser la méthode statique newInstance() pour obtenir une instance de la fabrique : elle détermine la classe à instancier en regardant dans l'ordre :

- Utilisation de la propriété système javax.xml.stream.XMLOutputFactory
- Utilisation du fichier lib/xml.stream.properties dans le répertoire d'installation du JRE
- Utilisation de l'API Services avec le fichier META-INF/services/javax.stream.XMLOutputFactory du jar
- Utilisation de l'instance par défaut

La classe XMLOuputFactory ne propose qu'une seule propriété :

Propriété	Rôle
javax.xml.stream.isRepairingNamespaces	Préciser si un préfixe par défaut doit être créé pour les espaces de nommage

La classe XMLEventFactory est une fabrique qui permet de créer des objets qui héritent de XMLEvent.

Il faut utiliser la méthode statique newInstance() pour obtenir une instance de la fabrique : elle détermine la classe à instancier en regardant dans l'ordre :

- Utilisation de la propriété système javax.xml.stream.XMLEventFactory
- Utilisation du fichier lib/xml.stream.properties dans le répertoire d'installation du JRE
- Utilisation de l'API Services avec le fichier META-INF/services/javax.stream.XMLEventFactory du jar
- Utilisation de l'instance par défaut

La classe XMLEventFactory ne possède pas de propriétés.

Pour modifier les propriétés de la fabrique, il faut utiliser la méthode setProperty() qui attend en paramètre le nom de la propriété et sa valeur.

Exemple :

```
XMLInputFactory xmlif = XMLInputFactory.newInstance();

xmlif.setProperty("javax.xml.stream.isCoalescing",Boolean.TRUE);
```

```
    xmlif.setProperty("javax.xml.stream.isReplacingEntityReferences", Boolean.TRUE);

    XMLStreamReader xmlsr = xmlif.createXMLStreamReader(new FileReader(
        "biblio.xml"));
```

Il est important de valoriser les propriétés avant de créer une instance d'un parseur. Une fois cette instance créée, il n'est plus possible de modifier ces propriétés.

Certains paramètres de configuration sont optionnels et ne sont donc pas obligatoirement supportés par une implémentation donnée. La méthode isPropertySupported() des fabriques XMLInputFactory et XMLOuputFactory permet de vérifier le support d'une propriété dont le nom est fourni en paramètre.

Exemple :

```
...
      XMLInputFactory xmlif =  XMLInputFactory.newInstance();
      if  (xmlif.isPropertySupported("javax.xml.stream.isReplacingEntityReferences"))  {
        System.out.println("javax.xml.stream.isReplacingEntityReferences  supporte");
        xmlif.setProperty("javax.xml.stream.isReplacingEntityReferences",  Boolean.TRUE);
      }
      XMLStreamReader xmlsr =  xmlif.createXMLStreamReader(new FileReader(
          "biblio.xml"));
  ...
```

XMLStreamReader et XMLEventReader possèdent la méthode getProperty() qui permet d'obtenir la valeur d'une propriété.

38.4. Le traitement d'un document XML avec l'API du type curseur

L'API de type curseur ne permet un parcours du document que vers l'avant : l'analyseur StAX parcours le flux de caractères du document et émet des événements à la demande.

L'interface principale de l'API de type curseur est XMLStreamReader : elle propose des méthodes pour le parcours du document et de nombreuses méthodes qu'il ne faut utiliser que dans le contexte de l'événement en cours de traitement. Les informations retournées par ces méthodes le sont sous la forme de chaînes de caractères directement extraites du document : ceci rend les traitements d'analyse peu consommateur en ressources.

Lors l'analyse d'un document, l'instance de l'interface XMLStreamReader permet de se déplacer dans les différents éléments qui composent le document XML en cours de traitement. Ce déplacement ne peut se faire que vers l'avant sans retour. Un événement est émis par le parseur à la demande de l'application : celui-ci correspond au type de l'élément courant dans le document.

Il est nécessaire de créer une instance de la classe XMLStreamReader en utilisant la fabrique XMLInputFactory. Il faut obtenir une instance de la fabrique XMLInputFactory en utilisant sa méthode newInstance().

Exemple :

```
XMLInputFactory xmlif = XMLInputFactory.newInstance();
```

Il faut instancier un objet de type XMLStreamReader en utilisant la méthode createXMLStreamReader() de la fabrique qui possède plusieurs surcharges acceptant en paramètre un objet de type Reader ou InputStream.

Exemple :

```
    XMLStreamReader xmlsr = xmlif.createXMLStreamReader(new FileReader(
        "biblio.xml"));
```

Il est alors possible d'itérer sur une demande au parseur de l'événement suivant grâce aux méthodes hasNext() et next() de la classe XMLStreamReader qui permettent de parcourir séquentiellement chaque événement émis par le parseur.

La méthode hasNext() renvoie un booléen qui précise si au moins un événement est encore disponible pour traitement. La méthode next() permet d'obtenir un identifiant sur l'événement suivant dans le flux de lecture du document.

Ceci permet d'itérer sur les événements jusqu'à ce qu'il n'y en ai plus à traiter en réalisant une itération tant que la méthode hasNext() renvoie true et d'appeler dans cette itération la méthode next().

Remarque : bien que les méthodes hasNext() et next() soient définies dans l'interface Iterator, l'interface XMLStreamReader n'hérite pas de cette interface.

La mise en oeuvre classique consiste donc à réaliser une itération sur les événements et à réaliser les traitements en fonction des événements.

Exemple :

```
int eventType;
while (xmlsr.hasNext()) {
  eventType = xmlsr.next();
  ...
}
```

La méthode next() renvoie un code sous la forme d'un entier qui précise le type d'événement qui a été rencontré lors de la lecture d'un élément du document. Ce code correspond à un type défini sous la forme de constante dans l'interface XMLStreamConstants.

Lors du parcours successif des éléments qui composent le document en cours de traitement, un événement particulier permet de déterminer le type d'éléments qui est en cours de traitement. Les événements qui peuvent être retournés par la méthode next() sont :

Evénement	Rôle
START_DOCUMENT	Le début du document (le prologue)
START_ELEMENT	Une balise ouvrante
ATTRIBUTE	Un attribut
NAMESPACE	La déclaration d'un espace de nommage
CHARACTERS	Du texte entre deux balises (pas forcément une balise ouvrante et sa balise fermante)
COMMENT	Un commentaire
SPACE	Un séparateur
PROCESSING_INSTRUCTION	Une instruction de traitement
DTD	Une DTD
ENTITY_REFERENCE	Une entité de référence
CDATA	Une section CData
END_ELEMENT	Une balise fermante
END_DOCUMENT	La fin du document
ENTITY_DECLARATION	La déclaration d'une entité
NOTATION_DECLARATION	La déclaration d'une notation

La méthode getEventType() permet de connaître le type de l'événement courant.

Il est nécessaire de traiter chaque événement en fonction des besoins : généralement un opérateur switch est utilisé pour définir les traitements de chaque événement utile.

Exemple :

```
      eventType = xmlsr.next();
      switch (eventType) {
      case XMLEvent.START_ELEMENT:
        System.out.println(xmlsr.getName());
        break;
      case XMLEvent.CHARACTERS:
        String chaine = xmlsr.getText();
        if (!xmlsr.isWhiteSpace()) {
          System.out.println("\t->\"" + chaine + "\"");
        }
        break;
      default:
        break;
      }
```

Le premier événement émis lors de l'analyse du document est de type START_DOCUMENT.

A chaque itération, les traitements peuvent traiter ou ignorer l'événement en fonction des besoins. Ceci permet d'avoir une grande liberté sur l'analyse du document.

Exemple :

```
package com.jmdoudoux.test.stax;

import java.io.FileReader;

import javax.xml.stream.XMLInputFactory;
import javax.xml.stream.XMLStreamReader;
import javax.xml.stream.events.XMLEvent;

public class TestStax1 {
  public static void main(String args[]) throws Exception {
    XMLInputFactory xmlif = XMLInputFactory.newInstance();
    XMLStreamReader xmlsr = xmlif.createXMLStreamReader(new FileReader(
        "biblio.xml"));
    int eventType;
    while (xmlsr.hasNext()) {
      eventType = xmlsr.next();
      switch (eventType) {
      case XMLEvent.START_ELEMENT:
        System.out.println(xmlsr.getName());
        break;
      case XMLEvent.CHARACTERS:
        String chaine = xmlsr.getText();
        if (!xmlsr.isWhiteSpace()) {
          System.out.println("\t->\""+chaine+"\"" );
        }
        break;
      default:
        break;
      }
    }
  }
}
```

Résultat :

```
{http://www.jmdoudoux.com/test/jaxb}bibliotheque
{http://www.jmdoudoux.com/test/jaxb}livre
{http://www.jmdoudoux.com/test/jaxb}titre
        ->"titre 1"
{http://www.jmdoudoux.com/test/jaxb}auteur
{http://www.jmdoudoux.com/test/jaxb}nom
        ->"nom 1"
```

```
{http://www.jmdoudoux.com/test/jaxb}prenom
        ->"prenom 1"
{http://www.jmdoudoux.com/test/jaxb}editeur
        ->"editeur 1"
{http://www.jmdoudoux.com/test/jaxb}livre
{http://www.jmdoudoux.com/test/jaxb}titre
        ->"titre 2"
{http://www.jmdoudoux.com/test/jaxb}auteur
{http://www.jmdoudoux.com/test/jaxb}nom
        ->"nom 2"
{http://www.jmdoudoux.com/test/jaxb}prenom
        ->"prenom 2"
{http://www.jmdoudoux.com/test/jaxb}editeur
        ->"editeur 2"
{http://www.jmdoudoux.com/test/jaxb}livre
{http://www.jmdoudoux.com/test/jaxb}titre
        ->"titre 3"
{http://www.jmdoudoux.com/test/jaxb}auteur
{http://www.jmdoudoux.com/test/jaxb}nom
        ->"nom 3"
{http://www.jmdoudoux.com/test/jaxb}prenom
        ->"prenom 3"
{http://www.jmdoudoux.com/test/jaxb}editeur
        ->"editeur 3"
```

L'interface XMLStreamReader propose des méthodes pour obtenir des données sur l'élément courant en fonction de l'événement lié à cet élément.

Plusieurs méthodes permettent d'obtenir des informations sur l'élément courant du curseur :
String getName();
String getLocalName();
String getNamespaceURI();
String getText();
String getElementText();
int getEventType();
Location getLocation();
int getAttributeCount();
QName getAttributeName(int);
String getAttributeValue(String, String);

Elle propose plusieurs méthodes pour obtenir des informations sur les attributs :
int getAttributeCount();
String getAttributeNamespace(int index);
String getAttributeLocalName(int index);
String getAttributePrefix(int index);
String getAttributeType(int index);
String getAttributeValue(int index);
String getAttributeValue(String namespaceUri,String localName);
boolean isAttributeSpecified(int index);

Elle propose plusieurs méthodes pour obtenir des informations sur les espaces de nommage :
int getNamespaceCount();
String getNamespacePrefix(int index);
String getNamespaceURI(int index);

Certaines méthodes sont utilisables selon l'événement pour obtenir un complément d'information sur l'entité courante correspondant à l'événement. Ces méthodes ne sont utilisables uniquement que dans un contexte précis. Par exemple, la méthode getAttributeValue() n'est utilisable que sur un événement de type START_ELEMENT.

Le tableau ci-dessous précise quelles sont les méthodes utilisables pour chaque événement :

Evénement	Méthodes
Tous	

	getProperty(), hasNext(), require(), close(), getNamespaceURI(), isStartElement(), isEndElement(), isCharacters(), isWhiteSpace(), getNamespaceContext(), getEventType(),getLocation(), hasText(), hasName()
START_ELEMENT	next(), getName(), getLocalName(), hasName(), getPrefix(), getAttributeXXX(), isAttributeSpecified(), getNamespaceXXX(), getElementText(), nextTag()
ATTRIBUTE	next(), nextTag(), getAttributeXXX(), isAttributeSpecified()
NAMESPACE	next(), nextTag(), getNamespaceXXX()
END_ELEMENT	next(), getName(), getLocalName(), hasName(), getPrefix(), getNamespaceXXX(), nextTag()
CHARACTERS	next(), getTextXXX(), nextTag()
CDATA	next(), getTextXXX(), nextTag()
COMMENT	next(), getTextXXX(), nextTag()
SPACE	next(), getTextXXX(), nextTag()
START_DOCUMENT	next(), getEncoding(), getVersion(), isStandalone(), standaloneSet(), getCharacterEncodingScheme(), nextTag()
END_DOCUMENT	close()
PROCESSING_INSTRUCTION	next(), getPITarget(), getPIData(), nextTag()
ENTITY_REFERENCE	next(), getLocalName(), getText(), nextTag()
DTD	next(), getText(), nextTag()

Il est préférable d'utiliser la méthode close() de la classe XMLStreamReader à la fin des traitements pour libérer les ressources.

Exemple :

```
    xmlsr.close();
```

L'exemple suivant propose un exemple complet.

Exemple : le document à traiter

```
<?xml version="1.0" encoding="UTF-8"?>
<tns:bibliotheque  xmlns:tns="http://www.jmdoudoux.com/test/stax" *
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.jmdoudoux.com/test/stax biblio2.xsd  ">
  <?MonTraitement?>
  <tns:livre>
      <!-- mon commentaire -->
    <tns:titre>titre 1</tns:titre>
    <tns:auteur>
      <tns:nom>nom 1</tns:nom>
      <tns:prenom>prenom  1</tns:prenom>
    </tns:auteur>
    <tns:editeur>editeur  1</tns:editeur>
  </tns:livre>
</tns:bibliotheque>
```

Exemple : parcours du document

```
package com.jmdoudoux.test.stax;

import java.io.FileReader;

import javax.xml.stream.XMLInputFactory;
import javax.xml.stream.XMLStreamReader;
```

```
import javax.xml.stream.events.XMLEvent;

public class TestStax6 {
  public static void main(String args[]) throws Exception {
    XMLInputFactory xmlif = XMLInputFactory.newInstance();
    XMLStreamReader xmlsr = xmlif.createXMLStreamReader(new FileReader(
        "biblio2.xml"));
    int eventType;
    while (xmlsr.hasNext()) {
      eventType = xmlsr.next();
      switch (eventType) {
      case XMLEvent.START_ELEMENT:
        System.out.println("START_ELEMENT : " + xmlsr.getName());
        break;
      case XMLEvent.START_DOCUMENT:
        System.out.println("START_DOCUMENT : " + xmlsr.getName());
        break;
      case XMLEvent.END_ELEMENT:
        System.out.println("END_ELEMENT : " + xmlsr.getName());
        break;
      case XMLEvent.END_DOCUMENT:
        System.out.println("END_DOCUMENT : ");
        break;
      case XMLEvent.COMMENT:
        System.out.println("COMMENT : "+ xmlsr.getText());
        break;
      case XMLEvent.CHARACTERS:
        System.out.println("CHARACTERS : ");
        break;
      case XMLEvent.PROCESSING_INSTRUCTION:
        System.out.println("PROCESSING_INSTRUCTION : "+ xmlsr.getPITarget());
        break;
      default:
        break;
      }
    }
  }
}
```

Résultat d'exécution :

```
START_ELEMENT : {http://www.jmdoudoux.com/test/stax}bibliotheque
CHARACTERS :
PROCESSING_INSTRUCTION : MonTraitement
CHARACTERS :
START_ELEMENT : {http://www.jmdoudoux.com/test/stax}livre
CHARACTERS :
COMMENT :  mon commentaire
CHARACTERS :
START_ELEMENT : {http://www.jmdoudoux.com/test/stax}titre
CHARACTERS :
END_ELEMENT : {http://www.jmdoudoux.com/test/stax}titre
CHARACTERS :
START_ELEMENT : {http://www.jmdoudoux.com/test/stax}auteur
CHARACTERS :
START_ELEMENT : {http://www.jmdoudoux.com/test/stax}nom
CHARACTERS :
END_ELEMENT : {http://www.jmdoudoux.com/test/stax}nom
CHARACTERS :
START_ELEMENT : {http://www.jmdoudoux.com/test/stax}prenom
CHARACTERS :
END_ELEMENT : {http://www.jmdoudoux.com/test/stax}prenom
CHARACTERS :
END_ELEMENT : {http://www.jmdoudoux.com/test/stax}auteur
CHARACTERS :
START_ELEMENT : {http://www.jmdoudoux.com/test/stax}editeur
CHARACTERS :
END_ELEMENT : {http://www.jmdoudoux.com/test/stax}editeur
CHARACTERS :
END_ELEMENT : {http://www.jmdoudoux.com/test/stax}livre
CHARACTERS :
END_ELEMENT : {http://www.jmdoudoux.com/test/stax}bibliotheque
END_DOCUMENT :
```

Chaque événement de type StartElement possède un événement correspondant de type EndElement même si le tag est sous sa forme réduite (<exemple/>)

Par défaut, les attributs n'émettent pas d'événement mais sont accessibles via une collection à partir de l'événement StartElement. Il en est de même avec les espaces de nommage.

Remarque : une partie texte du document peut émettre plusieurs événements de type Characters.

Durant le traitement du document, l'analyseur maintient une pile des espaces de nommage qui sont utilisés. Il est tout à fait possible d'interrompre le parcours du document : c'est un grand avantage de StAX de fournir le contrôle de la progression de l'analyse à l'application

Par exemple, si il n'est nécessaire de traiter qu'un seul tag, il suffit de tester la valeur du tag sur un événement de type START_ELEMENT, de réaliser les traitements sur le tag puis d'arrêter le parcours du document.

Exemple :

```
package com.jmdoudoux.test.stax;

import java.io.FileReader;

import javax.xml.stream.XMLInputFactory;
import javax.xml.stream.XMLStreamReader;
import javax.xml.stream.events.XMLEvent;

public class TestStax7 {
  public static void main(String args[]) throws Exception {
    XMLInputFactory xmlif = XMLInputFactory.newInstance();
    XMLStreamReader xmlsr = xmlif.createXMLStreamReader(new FileReader(
        "biblio.xml"));
    int eventType;
    boolean encore = xmlsr.hasNext();

    while (encore) {

      eventType = xmlsr.next();
      if (eventType == XMLEvent.START_ELEMENT) {
        System.out.println("element=" + xmlsr.getLocalName());
        if (xmlsr.getLocalName().equals("editeur")) {
          xmlsr.next();
          System.out.println("Premier editeur : " + xmlsr.getText());
          encore = false;
        }
      }
      if (!xmlsr.hasNext()) {
        encore = false;
      }
    }
  }
}
```

Résultat :

```
element=bibliotheque
element=livre
element=titre
element=auteur
element=nom
element=prenom
element=editeur
Premier editeur : editeur 1
```

38.5. Le traitement d'un document XML avec l'API du type itérateur

L'API de type itérateur repose sur l'interface XMLEventReader qui représente le parseur et sur l'interface XMLEvent qui représente un événement. Ces événements sont réutilisables et peuvent être enrichis avec des événements personnalisés.

L'interface XMLEventReader propose plusieurs méthodes pour itérer sur le document XML et obtenir l'événement courant.

Les événements émis lors de l'analyse du document sont encapsulés dans un objet de type XMLEvent qui possède pour chaque événement une classe fille : Attribute, Characters, Comment, StartDocument, EndDocument, StartElement, EndElement, Namespace, DTD, EntityDeclaration, EntityReference, NotationDeclaration, et ProcessingInstruction. Chacune de ces classes possède des propriétés dédiées.

L'API de type itérateur propose plusieurs types d'événements qui implémentent l'interface XMLEvent :

Evénement	Rôle
StartDocument	Concerne le début du document (le prologue)
StartElement	Concerne le début d'un élément
EndElement	Concerne la fin d'un élément
Characters	Concerne une section de type CData ou une entité de type CharacterEntities. Les séparateurs sont également représentés par cet événement
EntityReference	Concerne une entité de référence
ProcessingInstruction	Concerne une instruction de traitement
Comment	Concerne un tag de commentaires
EndDocument	Concerne la fin du document
DTD	Concerne les informations sur la DTD
Attribut	Normalement les attributs sont représentés par un événement StartElement mais ils peuvent être représentés par cet événement
Namespace	Normalement les espaces de nommages sont représentés par un événement StartElement mais ils peuvent être représentés par cet événement

Remarque : les événements DTD, EntityDeclaration, EntityReférence, NotationDeclaration et ProcessingInstruction ne sont levés que si une DTD est associée au document en cours de traitement.

L'interface StartElement qui hérite de l'interface XMLEvent propose plusieurs méthodes pour obtenir des informations sur les attributs et les espaces de nommage :

- Attribute getAttributeByName() : renvoie un attribut à partir de son nom
- Iterator getAttributes() : permet de parcourir tous les attributs de l'élément
- NamespaceContext getNamespaceContext : renvoie le contexte de l'espace de nommage
- Iterator getNamespaces : permet de parcourir tous les espaces de nommage de l'élément
- String getNamespaceURI : retourne la valeur d'un préfixe dans le contexte de l'élément

L'interface XMLEventReader analyse le document XML et émet des événements sous la forme d'un objet de type XMLEvent.

L'utilisation de XMLEventReader est similaire à celle de XMLStreamReader. XMLEventReader permet en plus de connaître le prochain événement grâce à la méthode peek() sans consommer l'événement ce qui permet d'anticiper sur les traitements.

L'interface XMLEventReader définit plusieurs méthodes :

Méthode	Rôle

XMLEvent nextEvent()	obtenir et consommer l'événement suivant (avance dans l'itération)
boolean hasNext()	préciser si il y a encore un événement à traiter
XMLEvent peek()	obtenir le prochain événement sans le consommer (l'itération ne bouge pas)
next()	avancer dans l'itération et renvoie le prochain événement
XMLEvent nextTag()	obtenir le prochain événement dans l'itération en ignorant les séparateurs pour renvoyer le prochain événement de type START_ELEMENT ou END_ELEMENT

Pour utiliser l'API de type itérateur, plusieurs étapes sont nécessaires.

Il faut obtenir une instance de la fabrique XMLInputFactory en utilisant sa méthode statique newInstance().

Exemple :

```
    XMLInputFactory xmlif = XMLInputFactory.newInstance();
```

Il faut instancier un objet de type XMLEventReader en utilisant la méthode createXMLStreamReader() de la fabrique qui possède plusieurs surcharges acceptant en paramètre un objet de type Reader ou InputStream.

Exemple :

```
    XMLEventReader xmler = xmlif.createXMLEventReader(new FileReader(
        "biblio.xml"));
```

La méthode hasNext() renvoie un booléen qui précise si au moins un événement est encore disponible. La méthode nextEvent() permet d'obtenir l'événement suivant. Il suffit de faire une itération tant que la méthode hasNext() renvoie true et d'appeler dans cette itération la méthode nextEvent() pour parcourir tout le document.

Exemple :

```
    XMLEvent event;
    while (xmler.hasNext()) {
      event = xmler.nextEvent();
      ...
    }
```

L'interface XMLEvent propose la méthode getEventType() pour connaître le type de l'événement. Elle propose aussi plusieurs méthodes :

- isXXX() pour chaque événement qui renvoie un booléen indiquant si l'événement est du type XXX.
- asXXX() pour chaque événement qui renvoie une instance de XXX correspondant à l'événement qui est du type XXX.

Exemple complet :

```
package com.jmdoudoux.test.stax;

import java.io.*;
import javax.xml.stream.*;
import javax.xml.stream.events.*;

public class TestStax2 {

  public static void main(String args[]) throws Exception {

    XMLInputFactory xmlif = XMLInputFactory.newInstance();
    XMLEventReader xmler = xmlif.createXMLEventReader(new FileReader(
        "biblio.xml"));
```

```
    XMLEvent event;
    while (xmler.hasNext()) {
      event = xmler.nextEvent();
      if (event.isStartElement()) {
        System.out.println(event.asStartElement().getName());
      } else if (event.isCharacters()) {
        if (!event.asCharacters().isWhiteSpace()) {
          System.out.println("\t>" + event.asCharacters().getData());
        }
      }
    }
  }
}
```

Résultat de l'éxécution :

```
{http://www.jmdoudoux.com/test/jaxb}bibliotheque
{http://www.jmdoudoux.com/test/jaxb}livre
{http://www.jmdoudoux.com/test/jaxb}titre
        >titre 1
{http://www.jmdoudoux.com/test/jaxb}auteur
{http://www.jmdoudoux.com/test/jaxb}nom
        >nom 1
{http://www.jmdoudoux.com/test/jaxb}prenom
        >prenom 1
{http://www.jmdoudoux.com/test/jaxb}editeur
        >editeur 1
{http://www.jmdoudoux.com/test/jaxb}livre
{http://www.jmdoudoux.com/test/jaxb}titre
        >titre 2
{http://www.jmdoudoux.com/test/jaxb}auteur
{http://www.jmdoudoux.com/test/jaxb}nom
        >nom 2
{http://www.jmdoudoux.com/test/jaxb}prenom
        >prenom 2
{http://www.jmdoudoux.com/test/jaxb}editeur
        >editeur 2
{http://www.jmdoudoux.com/test/jaxb}livre
{http://www.jmdoudoux.com/test/jaxb}titre
        >titre 3
{http://www.jmdoudoux.com/test/jaxb}auteur
{http://www.jmdoudoux.com/test/jaxb}nom
        >nom 3
{http://www.jmdoudoux.com/test/jaxb}prenom
        >prenom 3
{http://www.jmdoudoux.com/test/jaxb}editeur
        >editeur 3
```

Comme avec l'API de type curseur, il est possible avec l'API de type itérateur d'interrompre le traitement du document.

Exemple :

```
package com.jmdoudoux.test.stax;

import java.io.*;
import javax.xml.stream.*;
import javax.xml.stream.events.*;

public class TestStax3 {

  public static void main(String args[]) throws Exception {
    boolean termine = false;
    XMLInputFactory xmlif = XMLInputFactory.newInstance();
    FileReader fr = new FileReader("biblio.xml");
    XMLEventReader xmler = xmlif.createXMLEventReader(fr);
    XMLEvent event;
    termine = !xmler.hasNext();

    while (!termine) {
      event = xmler.nextEvent();
```

```
      if (event.isStartElement()) {
        if (event.asStartElement().getName().getLocalPart() == "editeur") {
          event = xmler.nextEvent();
          System.out.println("Premier editeur = "+event.asCharacters().getData());
          termine = true;
        }
      }
      if (!termine && !xmler.hasNext()) {
        termine = true;
      }
    }
      fr.close();
      xmler.close();
  }
}
```

Les événements sont émis dans l'ordre de rencontre des éléments lors du parcours du document par le parseur.

Si le document XML est syntaxiquement correct, alors chaque événement de type StartElement possède un événement de type EndElement correspondant.

38.6. La mise en oeuvre des filtres

StAX propose la mise en oeuvre de filtre pour n'obtenir que les événements désirés.

Pour l'API de type itérateur, l'interface EventFilter définit la méthode accept() qui attend en paramètre un objet de type XMLEvent et renvoie un booléen qui précise si cet événement doit être traité.

Exemple :

```
package com.jmdoudoux.test.stax;

import javax.xml.stream.EventFilter;
import javax.xml.stream.events.XMLEvent;

public class MonEventFilter implements EventFilter {

  public boolean accept(XMLEvent event) {

    if (event.isStartElement() || event.isEndElement())
      return true;
    else
      return false;
  }

}
```

Pour utiliser le filtre, il faut créer une instance de la classe XMLStreamReader en utilisant la méthode createFilteredReader() de la fabrique XMLInputFactory. Elle attend en paramètre l'instance de XMLStreamReader pour le traitement du document et le filtre.

Exemple :

```
package com.jmdoudoux.test.stax;

import java.io.FileReader;

import javax.xml.stream.XMLInputFactory;
import javax.xml.stream.XMLStreamReader;
import javax.xml.stream.events.XMLEvent;

public class TestStax11 {
  public static void main(String args[]) throws Exception {
    XMLInputFactory xmlif = XMLInputFactory.newInstance();
```

```
    XMLStreamReader xmlr = xmlif.createXMLStreamReader(
        new FileReader("biblio.xml"));

    XMLStreamReader xmlsr =
      xmlif.createFilteredReader(xmlr, new MonStreamFilter());

    while (xmlsr.hasNext()) {
      int eventType = xmlsr.next();
      switch (eventType) {
      case XMLEvent.START_ELEMENT:
        System.out.println("START_ELEMENT "+xmlsr.getName());
        break;
      case XMLEvent.END_ELEMENT:
        System.out.println("END_ELEMENT "+xmlsr.getName());
        break;
      default:
        System.out.println("AUTRE "+xmlsr.getName());
        break;
      }
    }
  }
}
```

Résultat :

```
START_ELEMENT {http://www.jmdoudoux.com/test/jaxb}livre
START_ELEMENT {http://www.jmdoudoux.com/test/jaxb}titre
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}titre
START_ELEMENT {http://www.jmdoudoux.com/test/jaxb}auteur
START_ELEMENT {http://www.jmdoudoux.com/test/jaxb}nom
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}nom
START_ELEMENT {http://www.jmdoudoux.com/test/jaxb}prenom
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}prenom
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}auteur
START_ELEMENT {http://www.jmdoudoux.com/test/jaxb}editeur
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}editeur
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}livre
START_ELEMENT {http://www.jmdoudoux.com/test/jaxb}livre
START_ELEMENT {http://www.jmdoudoux.com/test/jaxb}titre
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}titre
START_ELEMENT {http://www.jmdoudoux.com/test/jaxb}auteur
START_ELEMENT {http://www.jmdoudoux.com/test/jaxb}nom
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}nom
START_ELEMENT {http://www.jmdoudoux.com/test/jaxb}prenom
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}prenom
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}auteur
START_ELEMENT {http://www.jmdoudoux.com/test/jaxb}editeur
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}editeur
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}livre
START_ELEMENT {http://www.jmdoudoux.com/test/jaxb}livre
START_ELEMENT {http://www.jmdoudoux.com/test/jaxb}titre
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}titre
START_ELEMENT {http://www.jmdoudoux.com/test/jaxb}auteur
START_ELEMENT {http://www.jmdoudoux.com/test/jaxb}nom
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}nom
START_ELEMENT {http://www.jmdoudoux.com/test/jaxb}prenom
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}prenom
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}auteur
START_ELEMENT {http://www.jmdoudoux.com/test/jaxb}editeur
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}editeur
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}livre
END_ELEMENT {http://www.jmdoudoux.com/test/jaxb}bibliotheque
```

Pour l'API de type curseur, l'interface StreamFilter définit la méthode accept() qui attend en paramètre un objet de type XMLStreamReader et renvoie un booléen qui précise si l'événement courant doit être traité.

Exemple :

```
package com.jmdoudoux.test.stax;
```

```
import javax.xml.stream.StreamFilter;
import javax.xml.stream.XMLStreamReader;

public class MonStreamfilter implements StreamFilter {

  public boolean accept(XMLStreamReader reader) {
    if(reader.isStartElement() || reader.isEndElement())
      return true;
    else
      return false;
  }

}
```

Pour utiliser le filtre, il faut créer une instance de la classe XMLEventReader en utilisant la méthode createFilteredReader de la fabrique XMLInputFactory. Elle attend en paramètre l'instance de XMLEventReader pour le traitement du document et le filtre.

Exemple :

```
package com.jmdoudoux.test.stax;

import java.io.*;
import javax.xml.stream.*;
import javax.xml.stream.events.*;

public class TestStAX12 {

  public static void main(String args[]) throws Exception {

    XMLInputFactory xmlif = XMLInputFactory.newInstance();
    XMLEventReader xmlr = xmlif.createXMLEventReader(new FileReader(
        "biblio.xml"));
    XMLEventReader xmler = xmlif.createFilteredReader(xmlr,
        new MonEventFilter());

    XMLEvent event;
    while (xmler.hasNext()) {
      event = xmler.nextEvent();
      if (event.isStartElement()) {
        System.out.println("StartElement=" + event.asStartElement().getName());
      } else if (event.isEndElement()) {
        System.out.println("EndElement=" + event.asEndElement().getName());
      } else {
        System.out.println("Autre");
      }
    }
  }
}
```

Résultat :

```
StartElement={http://www.jmdoudoux.com/test/jaxb}bibliotheque
StartElement={http://www.jmdoudoux.com/test/jaxb}livre
StartElement={http://www.jmdoudoux.com/test/jaxb}titre
EndElement={http://www.jmdoudoux.com/test/jaxb}titre
StartElement={http://www.jmdoudoux.com/test/jaxb}auteur
StartElement={http://www.jmdoudoux.com/test/jaxb}nom
EndElement={http://www.jmdoudoux.com/test/jaxb}nom
StartElement={http://www.jmdoudoux.com/test/jaxb}prenom
EndElement={http://www.jmdoudoux.com/test/jaxb}prenom
EndElement={http://www.jmdoudoux.com/test/jaxb}auteur
StartElement={http://www.jmdoudoux.com/test/jaxb}editeur
EndElement={http://www.jmdoudoux.com/test/jaxb}editeur
EndElement={http://www.jmdoudoux.com/test/jaxb}livre
StartElement={http://www.jmdoudoux.com/test/jaxb}livre
StartElement={http://www.jmdoudoux.com/test/jaxb}titre
EndElement={http://www.jmdoudoux.com/test/jaxb}titre
StartElement={http://www.jmdoudoux.com/test/jaxb}auteur
```

```
StartElement={http://www.jmdoudoux.com/test/jaxb}nom
EndElement={http://www.jmdoudoux.com/test/jaxb}nom
StartElement={http://www.jmdoudoux.com/test/jaxb}prenom
EndElement={http://www.jmdoudoux.com/test/jaxb}prenom
EndElement={http://www.jmdoudoux.com/test/jaxb}auteur
StartElement={http://www.jmdoudoux.com/test/jaxb}editeur
EndElement={http://www.jmdoudoux.com/test/jaxb}editeur
EndElement={http://www.jmdoudoux.com/test/jaxb}livre
StartElement={http://www.jmdoudoux.com/test/jaxb}livre
StartElement={http://www.jmdoudoux.com/test/jaxb}titre
EndElement={http://www.jmdoudoux.com/test/jaxb}titre
StartElement={http://www.jmdoudoux.com/test/jaxb}auteur
StartElement={http://www.jmdoudoux.com/test/jaxb}nom
EndElement={http://www.jmdoudoux.com/test/jaxb}nom
StartElement={http://www.jmdoudoux.com/test/jaxb}prenom
EndElement={http://www.jmdoudoux.com/test/jaxb}prenom
EndElement={http://www.jmdoudoux.com/test/jaxb}auteur
StartElement={http://www.jmdoudoux.com/test/jaxb}editeur
EndElement={http://www.jmdoudoux.com/test/jaxb}editeur
EndElement={http://www.jmdoudoux.com/test/jaxb}livre
EndElement={http://www.jmdoudoux.com/test/jaxb}bibliotheque
```

38.7. L'écriture un document XML avec l'API de type curseur

L'interface XMLStreamWriter propose des fonctionnalités simples et de bas niveau pour écrire un document.

L'interface XMLStreamWriter définit les méthodes pour un objet capable de réécrire un document en cours de parcours ou d'écrire un nouveau document.

Une instance d'un tel objet est obtenue en utilisant la fabrique XMLOutputFactory.

Exemple :

```
    XMLStreamWriter writer = XMLOutputFactory.newInstance().
        createXMLStreamWriter(outStream);
```

L'interface XMLStreamWriter propose de nombreuses méthodes pour ajouter des noeuds de différents types au document en cours de rédaction :

Méthode	Rôle
writeStartDocument()	ajouter le prologue du document
writeEndDocument()	ajouter tous les éléments de type fin requis pour terminer le document
writeStartElement()	ajouter un élément de type début
writeEndElement()	ajouter un élément de type fin
writeComment()	ajouter un élément de type commentaire
writeNamespace()	ajouter un espace de nommage
writeCharacters()	ajouter un élément de type texte
writeProcessingInstruction()	ajouter une instruction de traitement

Remarque : chaque méthode writeStartXxx() doit avoir un appel à la méthode writeEndXxx() correspondante dans les traitements.

Il faut obtenir une instance de la fabrique XMLOuputFactory utilisant sa méthode newInstance().

Exemple :

```
XMLOutputFactory outputFactory = XMLOutputFactory.newInstance();
```

Il faut instancier un objet de type FileWriter qui va encapsuler le fichier où sera stocké le document XML

Exemple :

```
FileWriter output = new FileWriter(new File("test.xml"));
```

Il faut obtenir une instance de l'interface XMLStreamWriter en utilisant la méthode createXMLStreamWriter() de la fabrique.

Exemple :

```
XMLStreamWriter xmlsw = outputFactory.createXMLStreamWriter(output);
```

Il faut créer le prologue du document en utilisant la méthode writeStartDocument() qui attend en paramètre le nom du jeu de caractères d'encodage et la version de xml. Ces deux informations ne sont utilisées que comme valeur des attributs encoding et version du prologue.

Exemple :

```
xmlsw.writeStartDocument("Cp1252", "1.0");
```

Pour préciser le jeu de caractères utilisé pour encoder le document XML, il est nécessaire d'utiliser une version surchargée de la méthode createXMLStreamWriter().

Exemple :

```
FileOutputStream output = new FileOutputStream("test.xml");
XMLStreamWriter xmlsw = outputFactory.createXMLStreamWriter(output, "UTF-8");
xmlsw.writeStartDocument("UTF-8", "1.0");
```

La création d'une balise dans le document à la position courante se fait en utilisant la méthode writeStartElement(). Cette méthode possède trois surcharges qui permettent de préciser le nom de la balise, son préfixe et l'URI de son espace de nommage.

La méthode writeNamespace() qui attend en paramètre un préfixe et une uri permet de définir un espace de nommage pour la balise courante.

Exemple :

```
xmlsw.writeNamespace("tns", "http://www.jmdoudoux.com/test/stax");
```

La méthode writeAttribut() permet de définir un attribut à la balise courante. Elle possède plusieurs surcharges qui attendent en paramètres le nom de l'attribut, sa valeur, un préfixe et l'uri de l'espace de nommage

Exemple :

```
xmlsw.writeAttribut("xsi", "http://www.w3.org/2001/XMLSchema-instance");
```

La méthode writeCharacters() qui peut être utilisée avec une chaîne de caractères ou un tableau de caractères permet d'écrire un noeud de type texte dans la balise courante.

Exemple :

```
      xmlsw.writeCharacters("titre "+i);
```

La méthode writeCharacters() permet d'ajouter du texte dans le document en échappant les caractères utilisés par XML (<, >, &, ...).

La méthode writeEndElement() permet de créer un balise fermante à la balise courante. Elle détermine automatiquement le nom de la balise courante pour créer la balise nécessaire. Son appel est obligatoire pour chaque balise ouverte.

Exemple :

```
      xmlsw.writeEndElement();
```

Une balise de commentaires peut être créée en utilisant la méthode writeComment().

Exemple :

```
    xmlsw.writeComment("Fichier de test XMLStreamWriter");
```

La méthode writeProcessingInstruction() permet d'ajouter une balise de type instruction de traitement.

La méthode writeEndDocument() permet de créer toutes les balises fermantes requises à partir de la balise courante jusqu'à la balise racine.

Une fois le document complet, il est nécessaire d'utiliser les méthodes flush() et close() de la classe XMLStreamWriter pour enregistrer le document XML dans le fichier.

Exemple :

```
    xmlsw.flush();
    xmlsw.close();
```

Exemple complet :

```
package com.jmdoudoux.test.stax;

import java.io.StringWriter;

import javax.xml.stream.XMLOutputFactory;
import javax.xml.stream.XMLStreamWriter;

public class TestStax5 {

  public static void main(String args[]) throws Exception {

    String ns = "http://www.jmdoudoux.com/test/stax";

    StringWriter strw = new StringWriter();
    XMLOutputFactory output = XMLOutputFactory.newInstance();
    XMLStreamWriter writer = output.createXMLStreamWriter(strw);
    writer.writeStartDocument();
    writer.setPrefix("tns",ns);
    writer.setDefaultNamespace(ns);
    writer.writeStartElement(ns,"bibliotheque");
      writer.writeNamespace("tns",ns);
      writer.writeStartElement(ns,"livre");
        writer.writeAttribute("id","1");
        writer.writeStartElement(ns,"titre");
          writer.writeCharacters("titre1");
        writer.writeEndElement();
        writer.writeStartElement(ns,"auteur");
          writer.writeStartElement(ns, "nom");
          writer.writeCharacters("nom1");
```

```
          writer.writeEndElement();
          writer.writeStartElement(ns,"prenom");
          writer.writeCharacters("prenom1");
          writer.writeEndElement();
        writer.writeEndElement();
        writer.writeStartElement(ns,"editeur");
        writer.writeCharacters("editeur1");
        writer.writeEndElement();
      writer.writeEndElement();
    writer.writeEndElement();
    writer.flush();

    System.out.println(strw.toString());
  }
}
```

Remarque : l'indentation des méthodes writeXxx() permet de vérifier qu'aucun appel de méthode n'a été oublié.

Résultat :

```
  <?xml version="1.0" ?>
  <bibliotheque  xmlns:tns="http://www.jmdoudoux.com/test/stax">
     <tns:livre id="1">
       <tns:titre>titre1</tns:titre>
       <tns:auteur>
         <tns:nom>nom1</nom>
         <tns:prenom>prenom1</tns:prenom>
       </tns:auteur>
       <tns:editeur>editeur1</tns:editeur>
    </tns:livre>
  </bibliotheque>
```

Attention : une implémentation de l'interface XMLStreamWriter n'a pas d'obligation de vérifier que le document créé soit bien formé. Par exemple, l'oubli d'un appel à la méthode writeEndElement() pour un tag provoque un décalage dans la balise de fin qui va résulter en l'absence de la balise de fermeture du tag racine.

Exemple complet :

```
package com.jmdoudoux.test.stax;

import java.io.File;
import java.io.FileWriter;

import javax.xml.stream.XMLOutputFactory;
import javax.xml.stream.XMLStreamWriter;

public class TestStax4 {

  public static void main(String args[]) throws Exception {

    XMLOutputFactory outputFactory = XMLOutputFactory.newInstance();
    FileWriter output = new FileWriter(new File("test.xml"));
    XMLStreamWriter xmlsw = outputFactory.createXMLStreamWriter(output);
    xmlsw.writeStartDocument("Cp1252", "1.0");
    xmlsw.writeComment("Fichier de test XMLStreamWriter");
    xmlsw.writeStartElement("tns", "bibliotheque",
        "http://www.jmdoudoux.com/test/stax");
    xmlsw.writeNamespace("tns", "http://www.jmdoudoux.com/test/stax");
    xmlsw.writeNamespace("xsi", "http://www.w3.org/2001/XMLSchema-instance");
    xmlsw.writeAttribute("xsi:schemaLocation",
        "http://www.jmdoudoux.com/test/stax/biblio.xsd");

    for (int i = 1; i > 4; i++) {

      xmlsw.writeStartElement("tns", "livre",
          "http://www.jmdoudoux.com/test/stax");

      xmlsw.writeStartElement("tns", "titre",
```

```
          "http://www.jmdoudoux.com/test/stax");
      xmlsw.writeCharacters("titre "+i);
      xmlsw.writeEndElement();

      xmlsw.writeStartElement("tns", "auteur",
          "http://www.jmdoudoux.com/test/stax");
      xmlsw.writeStartElement("tns", "nom",
          "http://www.jmdoudoux.com/test/stax");
      xmlsw.writeCharacters("nom "+i);
      xmlsw.writeEndElement();
      xmlsw.writeStartElement("tns", "prenom",
          "http://www.jmdoudoux.com/test/stax");
      xmlsw.writeCharacters("prenom "+i);
      xmlsw.writeEndElement();
      xmlsw.writeEndElement();

      xmlsw.writeStartElement("tns", "editeur",
          "http://www.jmdoudoux.com/test/stax");
      xmlsw.writeCharacters("editeur "+i);
      xmlsw.writeEndElement();

      xmlsw.writeEndElement();
    }

    xmlsw.writeEndElement();
    xmlsw.flush();
    xmlsw.close();

  }
}
```

Résultat :

```
  <?xml  version="1.0" encoding="Cp1252"?><!--Fichier de test  XMLStreamWriter-->
  <tns:bibliotheque  xmlns:tns="http://www.jmdoudoux.com/test/stax"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://www.jmdoudoux.com/test/stax/biblio.xsd">
    <tns:livre>
       <tns:titre>titre 1</tns:titre>
       <tns:auteur>
         <tns:nom>nom 1</tns:nom>
         <tns:prenom>prenom 1</tns:prenom>
       </tns:auteur>
       <tns:editeur>editeur 1</tns:editeur>
     </tns:livre>
     <tns:livre>
       <tns:titre>titre 2</tns:titre>
       <tns:auteur>
         <tns:nom>nom 2</tns:nom>
         <tns:prenom>prenom 2</tns:prenom>
       </tns:auteur>
       <tns:editeur>editeur 2</tns:editeur>
     </tns:livre>
     <tns:livre>
       <tns:titre>titre 3</tns:titre>
       <tns:auteur>
         <tns:nom>nom 3</tns:nom>
         <tns:prenom>prenom 3</tns:prenom>
       </tns:auteur>
       <tns:editeur>editeur 3</tns:editeur>
     </tns:livre>
  </tns:bibliotheque>
```

38.8. L'écriture un document XML avec l'API de type itérateur

L'interface XMLEventWriter propose des fonctionnalités à l'API de type itérateur pour écrire un document XML : celle-ci est particulièrement adaptée à la réécriture d'un document en cours de traitement par l'API de type itérateur mais

elle peut aussi être utilisée pour créer un nouveau document. Elle propose des méthodes pour créer un document XML à partir d'objets de type XMLEvent.

Une instance de type XMLEventWriter est obtenue en utilisant la fabrique XMLOutPutFactory.

Elle possède plusieurs méthodes dont les principales sont :

Méthode	Rôle
void flush()	Permettre de vider le cache et d'écrire les données qu'il contient
void close()	Fermer le flux d'écriture
void add(XMLEvent)	Ajouter un élément dans le document

Les événements sont ajoutés au fur et à mesure et ne peuvent plus être modifiés une fois ajoutés. L'ajout d'attributs ou d'espace de nommage se fait toujours sur le dernier élément de type StartElement ajouté dans le flux.

La méthode setPrefix() permet d'associer un préfixe à un espace de nommage.

Il faut instancier une occurrence de l'interface XMLEventWriter à partir d'une fabrique de type XMLOutputFactory.

Exemple :

```
    XMLOutputFactory outputFactory = XMLOutputFactory.newInstance();
    XMLEventWriter writer = outputFactory.createXMLEventWriter(new FileWriter(
        "test2.xml"));
```

Il faut obtenir une instance de la fabrique XMLEventFactory en utilisant sa méthode newInstance().

Exemple :

```
    XMLEventFactory eventFactory = XMLEventFactory.newInstance();
```

Cette fabrique permet de créer des instances des événements qui seront ajoutés dans le document.

Exemple :

```
    writer.add(eventFactory.createStartDocument());
```

Une fois le document terminé, il suffit d'appeler les méthodes flush() et close().

Exemple :

```
package com.jmdoudoux.test.stax;

import java.io.FileWriter;

import javax.xml.stream.XMLEventFactory;
import javax.xml.stream.XMLEventWriter;
import javax.xml.stream.XMLOutputFactory;

public class TestStax8 {

  private static final String NS_TNS = "http://www.jmdoudoux.com/test/stax";

  private static final String PREFIX_TNS = "tns";

  public static void main(String args[]) throws Exception {
    XMLOutputFactory outputFactory = XMLOutputFactory.newInstance();
    XMLEventWriter writer = outputFactory.createXMLEventWriter(new FileWriter(
        "test2.xml"));
```

```
    XMLEventFactory eventFactory = XMLEventFactory.newInstance();

    writer.setPrefix(PREFIX_TNS, NS_TNS);
    writer.add(eventFactory.createStartDocument());
    writer.add(eventFactory.createStartElement(PREFIX_TNS, NS_TNS,
        "bibliotheque"));
    writer.add(eventFactory.createNamespace(PREFIX_TNS, NS_TNS));
    writer.add(eventFactory.createProcessingInstruction("MonTraitement", ""));
    writer.add(eventFactory.createStartElement(PREFIX_TNS, NS_TNS, "livre"));
    writer.add(eventFactory.createComment("mon commentaire"));
    writer.add(eventFactory.createStartElement(PREFIX_TNS, NS_TNS, "titre"));
    writer.add(eventFactory.createCharacters("titre 1"));
    writer.add(eventFactory.createEndElement(PREFIX_TNS, NS_TNS, "titre"));
    writer.add(eventFactory.createStartElement(PREFIX_TNS, NS_TNS, "auteur"));
    writer.add(eventFactory.createStartElement(PREFIX_TNS, NS_TNS, "nom"));
    writer.add(eventFactory.createCharacters("nom 1"));
    writer.add(eventFactory.createEndElement(PREFIX_TNS, NS_TNS, "nom"));
    writer.add(eventFactory.createStartElement(PREFIX_TNS, NS_TNS, "prenom"));
    writer.add(eventFactory.createCharacters("prenom 1"));
    writer.add(eventFactory.createEndElement(PREFIX_TNS, NS_TNS, "prenom"));
    writer.add(eventFactory.createEndElement(PREFIX_TNS, NS_TNS, "auteur"));
    writer.add(eventFactory.createStartElement(PREFIX_TNS, NS_TNS, "editeur"));
    writer.add(eventFactory.createCharacters("editeur 1"));
    writer.add(eventFactory.createEndElement(PREFIX_TNS, NS_TNS, "editeur"));
    writer.add(eventFactory.createEndElement(PREFIX_TNS, NS_TNS, "livre"));
    writer.add(eventFactory
        .createEndElement(PREFIX_TNS, NS_TNS, "bibliotheque"));

    writer.add(eventFactory.createEndDocument());
    writer.flush();
    writer.close();

  }
}
```

Résultat :

```
<?xml version="1.0"?>
<tns:bibliotheque  xmlns:tns="http://www.jmdoudoux.com/test/stax">
  <?MonTraitement  ?>
    <tns:livre>
      <!--mon commentaire-->
      <tns:titre>titre  1</tns:titre>
      <tns:auteur>
        <tns:nom>nom  1</tns:nom>
        <tns:prenom>prenom  1</tns:prenom>
      </tns:auteur>
    <tns:editeur>editeur  1</tns:editeur>
  </tns:livre>>/p<
</tns:bibliotheque>
```

Il est aussi possible d'utiliser l'API de type itérateur en lecture et en écriture simultanément.

Exemple :

```
package com.jmdoudoux.test.stax;

import java.io.FileReader;
import java.io.FileWriter;

import javax.xml.stream.XMLEventFactory;
import javax.xml.stream.XMLEventReader;
import javax.xml.stream.XMLEventWriter;
import javax.xml.stream.XMLInputFactory;
import javax.xml.stream.XMLOutputFactory;
import javax.xml.stream.events.Characters;
import javax.xml.stream.events.XMLEvent;

public class TestStax9 {
```

```
  public static void main(String args[]) throws Exception {
    XMLOutputFactory outputFactory = XMLOutputFactory.newInstance();

    XMLInputFactory xmlif = XMLInputFactory.newInstance();
    FileReader fr = new FileReader("biblio.xml");
    XMLEventReader reader = xmlif.createXMLEventReader(fr);

    XMLEventFactory eventFactory = XMLEventFactory.newInstance();
    XMLEventWriter writer = outputFactory.createXMLEventWriter(new FileWriter(
        "test3.xml"));

    while (reader.hasNext()) {
      XMLEvent event = (XMLEvent) reader.next();
      if (event.getEventType() == XMLEvent.CHARACTERS) {
        Characters characters = event.asCharacters();
        if (!characters.isWhiteSpace()) {
          writer.add(eventFactory.createCharacters(characters.getData() + " modif"));
        }
      } else {
        writer.add(event);
      }
    }
    writer.flush();

    writer.close();

  }
}
```

Résultat :

```
<?xml  version="1.0"?>
  <tns:bibliotheque  xmlns:tns="http://www.jmdoudoux.com/test/jaxb"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.jmdoudoux.com/test/jaxb  biblio.xsd ">
     <tns:livre>
       <tns:titre>titre 1 modif</tns:titre>
       <tns:auteur>
         <tns:nom>nom 1 modif</tns:nom>
         <tns:prenom>prenom 1 modif</tns:prenom>
       </tns:auteur>
       <tns:editeur>editeur 1 modif</tns:editeur>
     </tns:livre>
     <tns:livre>
       <tns:titre>titre 2 modif</tns:titre>
       <tns:auteur>
         <tns:nom>nom 2 modif</tns:nom>
         <tns:prenom>prenom 2 modif</tns:prenom>
       </tns:auteur>
       <tns:editeur>editeur 2 modif</tns:editeur>
     </tns:livre>
     <tns:livre>
       <tns:titre>titre 3 modif</tns:titre>
       <tns:auteur>
         <tns:nom>nom 3 modif</tns:nom>
         <tns:prenom>prenom 3 modif</tns:prenom>
       </tns:auteur>
       <tns:editeur>editeur 3 modif</tns:editeur>
     </tns:livre>
</tns:bibliotheque>
```

38.9. La comparaison entre SAX, DOM et StAX

Les parseurs avant l'arrivée de StAX utilisent deux méthodes principales pour traiter un document XML :

- ceux basés sur un modèle événementiel utilisé par SAX notamment
- ceux basés sur un modèle reposant sur un arbre d'objets utilisé par DOM notamment

Ces deux modèles ont chacun leurs avantages et leurs inconvénients.

	Avantages	Inconvénients
SAX	• grande efficacité • faible ressource nécessaire • API simple • traitement au fur et à mesure de la lecture	• l'état du document doit être conservé « manuellement » • le document tout entier doit être parcouru • ne peut être utilisé que pour lire un document • traitements séquentiels
DOM	• lecture aléatoire dans l'arbre du document • permet la mise à jour d'un document	• ressources nécessaires importantes proportionnelles à la taille du document • API plus complexe car non développée spécifiquement pour Java

Les trois API de JAXP permettant d'analyser un document XML ont chacune des points forts et des points faibles dont il faut tenir compte pour déterminer quelle API sera la mieux adaptée en fonction des besoins.

	SAX	StAX	DOM
Type de traitements	Événement de type push	Événement de type pull	Arbre d'objets en mémoire
Facilité de mise en oeuvre	Moyenne	Elevée	Moyenne
Support XPath	Non	Non	Oui
Consommation en ressources	Faible	Faible	Dépendante de la taille du document
Sens de parcours	Vers l'avant uniquement	Vers l'avant uniquement	Libre
Lecture	Oui	Oui	Oui
Ecriture	Non	Oui	Oui
Modification	Non	Non	Oui

Même si l'API StAX est basée sur des événements, ces fonctionnalités la placent entre les deux autres types de parsers.

SAX et StAX reposent tous les deux sur un traitement par flux : le document est parcouru et traité au fur et à mesure. Ce type de traitement est efficace et peu consommateur en ressources : il est donc particulièrement adapté au traitement de gros documents.

L'avantage de StAX par rapport à SAX est de donner la possibilité au développeur de demander le prochain événement et de le traiter si nécessaire plutôt que de fournir des traitements dans des fonctions de type « callback » appelées par le parseur. Ceci donne au développeur un meilleur contrôle sur les traitements en facilitant leur mise en oeuvre et permet à tout moment d'interrompre le traitement du parseur sans attendre le traitement de tout le document.

SAX lit et analyse le document au fur et à mesure et émet des événements à destination d'un handler définit dans l'application qui est composée de méthodes de type callback. Ces méthodes sont automatiquement exécutées par le parseur en fonction des événements émis par ce dernier lors de la lecture du document. C'est donc le parseur qui a le contrôle sur les traitements d'analyse du document : ce type de traitement est dit push (c'est le parseur qui émet des événements à son initiative vers l'application). Il nécessite le parcours de tout le document. SAX ne permet d'écrire un document XML.

SAX n'est pas aussi simple à mettre en oeuvre que StAX puisqu'il faut développer un handler qui va traiter les événements émis sous la forme de callback : le code des traitements pour sa mise en oeuvre peut être rapidement complexe. Stax est plus simple que SAX : c'est une forme de traitement de type pull (les événements sont émis par le parseur à la demande de l'application) ce qui permet de donner le contrôle de l'analyse au développeur grâce à un parcours d'un ensemble d'événements.

Avec StAX, c'est donc l'application qui possède le contrôle sur le traitement du document ce qui rend plus intuitif le code à écrire pour traiter le document : l'application peut ignorer un élément, appliquer un filtre ou arrêter le traitement du document à tout moment.

Les fonctionnalités de StAX sont proches de celles de SAX. Cependant StAX propose des fonctionnalités supplémentaires :

- une mise en oeuvre des traitements sous une forme itérative qui la rend plus naturelle que la forme de type callback de SAX
- StAX permet l'écriture de documents
- StAX peut être plus efficace car il n'oblige pas à traiter tout le document

StAX est donc aussi efficace que SAX en proposant un modèle de mise en oeuvre plus facile et extensible. StAX pourrait remplacer SAX mais StAX est une API récente qui ne possède pas d'implémentation dans d'autre langage pour le moment. SAX est un standard de fait implémenté dans de nombreuses solutions de parsing dans différentes plates-formes et langages.

DOM est basé sur un arbre d'objets en mémoire qui représente l'ensemble des éléments d'un document XML. Ceci est très pratique pour permettre de se déplacer librement dans le document et de le parcourir à son gré d'autant que DOM supporte l'utilisation des expressions XPath.

DOM est la seule API qui permet de modifier le document. La contre partie est que DOM consomme beaucoup de ressources et notamment de mémoire puisque tout le document est représenté par un arbre d'objets en mémoire : cela exclut de fait son utilisation pour des documents XML volumineux.

DOM est donc l'API la plus puissante puisqu'elle permet un parcours du document dans n'importe quel ordre et quel sens, de modifier le document (création, modification et suppression de noeuds dans le document) et d'écrire le document.

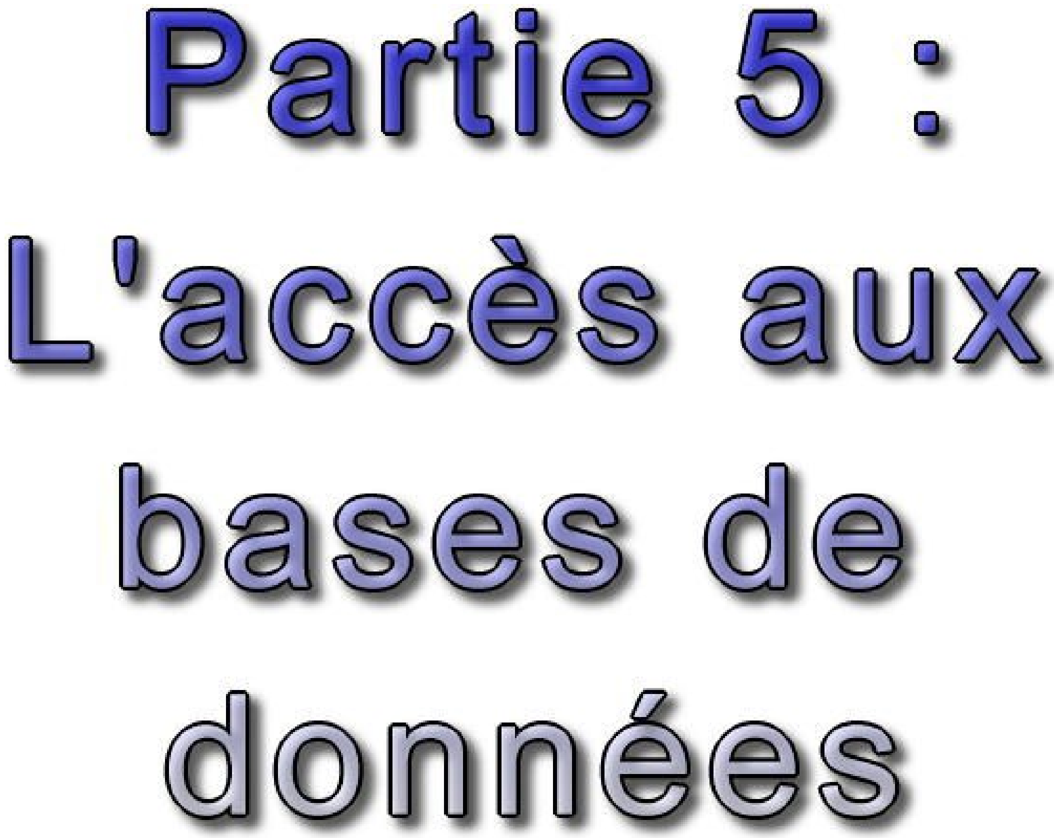
DOM et StAX ont en commun de pouvoir écrire un document XML.

L'existence de trois API pour traiter un document XML entraîne logiquement des interrogations sur le choix de l'API à utiliser en fonction du besoin. Il n'existe pas de règles immuables concernant ces choix mais voici quelques cas d'utilisation particuliers :

- La transformation d'un document XML en un autre document XML : il est fréquent de devoir transformer un document XML en un autre document XML pour modifier la structure du document ou pour enrichir ou appauvrir les données contenues dans le document. La solution la plus adaptée pour modifier la structure ou appauvrir le document semble être XSLT puisque c'est son rôle principal. StAX ou DOM peuvent être aussi utilisés notamment dans le cas d'enrichissement du document : ces deux API nécessitent l'écriture de code mais cela permet aussi un accès à toutes les API de Java.
- Data Binding : l'utilisation de plus en plus fréquente de XML nécessite de pouvoir mapper un objet à un document ou une portion de document XML et vice et versa. Le plus simple est d'utiliser une API dédiée telle que JAXB mais il est aussi possible de réaliser ce traitement à la main. SAX ne peut être utilisé que pour mapper un document XML dans un objet (unmarshalling). StAX et DOM pouvant écrire un document, ces deux API peuvent être utilisés pour des opérations dans les deux sens.
- Un document XML comme source de données : les données à utiliser et éventuellement à mettre à jour sont stockées dans un document XML. Dans ce cas de figure, seul DOM peut répondre au besoin grâce à son support de XPath pour accéder directement à une donnée et sa possibilité de modifier le contenu du document pour réaliser des mises à jour.

StAX peut donc être utilisé dans de nombreux cas de traitements de documents XML.

Partie 5 : L'accès aux bases de données



Cette partie concerne l'accès aux bases de données à partir d'applications Java. Il existe pour ce besoin de nombreuses solutions sous différentes formes dont plusieurs API standards ou frameworks open source.

Cette partie regroupe plusieurs chapitres :

- La persistance des objets : expose les difficultés liées à la persistance des objets vis à vis du modèle relationnel et présente rapidement des solutions architecturales et techniques (api standards et open source)
- JDBC (Java DataBase Connectivity) : indique comment utiliser cette API historique pour accéder aux bases de données
- JDO (Java Data Object) : API qui standardise et automatise le mapping en des objets Java et un système de gestion de données
- Hibernate : présente Hibernate, un framework de mapping Objets/Relationnel open source
- JPA (Java Persistence API) : JPA est la spécification de l'API standard dans le domaine du mapping O/R utilisable avec Java EE mais aussi avec Java SE à partir de la version 5.

39. La persistance des objets

39.1. Introduction

La quasi-totalité des applications de gestion traitent des données dans des volumes plus ou moins important. Dès que ce volume devient assez important, les données sont stockées dans une base de données.

Il existe plusieurs types de base de données

- Hiérarchique : historiquement le type le plus ancien, ces bases de données étaient largement utilisées sur les gros systèmes de type mainframe. Les données sont organisées de façon hiérarchique grâce à des pointeurs. Exemple DL1, IMS, Adabas
- Relationnelle (RDBMS / SGBDR) : c'est le modèle le plus répandu actuellement. Ce type de base de données repose sur les théories ensemblistes et l'algèbre relationnel. Les données sont organisées en tables possédant des relations entre elles grâce à des clés primaires et étrangères. Les opérations sur la base sont réalisées grâce à des requêtes SQL. Exemple : MySQL, PosgreSQL, HSQLDB, Derby
- Objet (ODBMS / SGBDO) : Exemple db4objects
- XML (XDBMS) : Exemple : Xindice

La seconde catégorie est historiquement la plus répandue mais aussi une des moins compatible avec la programmation orienté objet.

39.1.1. La correspondance entre le modèle relationnel et objet

La correspondance des données entre le modèle relationnel et le modèle objet doit faire face à plusieurs problèmes :

- le modèle objet propose plus de fonctionnalités : héritage, polymorphisme, ...
- les relations entre les entités des deux modèles sont différentes
- les objets ne possèdent pas d'identifiant en standard (hormis son adresse mémoire qui varie d'une exécution à l'autre). Dans le modèle relationnel, chaque occurrence devrait posséder un identifiant unique

La persistance des objets en Java possède de surcroît quelques inconvénients supplémentaires :

- de multiples choix dans les solutions et les outils (standard, commerciaux, open source)
- de multiple choix dans les API et leurs implémentations
- de nombreuses évolutions dans les API standards et les frameworks open source

39.2. L'évolution des solutions de persistance avec Java

La première approche pour faire une correspondance entre ces deux modèles a été d'utiliser l'API JDBC fournie en standard avec le JDK. Cependant cette approche possède plusieurs inconvénients majeurs :

- nécessite l'écriture de nombreuses lignes de codes, souvent répétitives
- le mapping entre les tables et les objets est un travail de bas niveau
- ...

Tous ces facteurs réduisent la productivité mais aussi les possibilités d'évolutions et de maintenance. De plus, une grande partie de ce travail peut être automatisé.

Face à ce constat, différentes solutions sont apparues :

- des frameworks open source : le plus populaire est Hibernate qui utilise des POJOs.
- des frameworks commerciaux dont Toplink était le leader avant que sa base passe en open source
- des API Standards : JDO, EJB entity, JPA

39.3. Le mapping O/R (objet/relationnel)

Le mapping Objet/Relationnel (mapping O/R) consiste à réaliser la correspondance entre le modèle de données relationnel et le modèle objets de façon la plus facile possible.

Un outil de mappping O/R doit cependant proposer un certain nombre de fonctionnalités parmi lesquelles :

- Assurer le mapping des tables avec les classes, des champs avec les attributs, des relations et des cardinalités
- Proposer une interface qui permette de facilement mettre en oeuvre des actions de type CRUD
- Eventuellement permettre l'héritage des mappings
- Proposer un langage de requêtes indépendant de la base de données cible et assurer une traduction en SQL natif selon la base utilisée
- Supporter différentes formes d'identifiants générés automatiquement par les bases de données (identity, sequence, ...)
- Proposer un support des transactions
- Assurer une gestion des accès concurrents (verrou, dead lock, ...)
- Fournir des fonctionnalités pour améliorer les performances (cache, lazy loading, ...)

Les solutions de mapping sont donc riches en fonctionnalités ce qui peut rendre leur mise en oeuvre plus ou moins complexe. Cette complexité est cependant différente d'un développement de toute pièce avec JDBC.

Les solutions de mapping O/R permettent de réduire la quantité de code à produire mais impliquent une partie configuration (généralement sous la forme d'un ou plusieurs fichiers XML ou d'annotations pour les solutions reposant sur Java 5).

Depuis quelques années, les principales solutions mettent en oeuvre des POJO (Plain Old Java Object).

39.3.1. Le choix d'une solution de mapping O/R

Comme pour tout choix d'une solution, des critères standards doivent entrer en ligne de compte (prix, complexité de prise en main, performance, maturité, pérennité, support, ...)

Dans le cas d'une solution de mapping O/R, il faut aussi prendre en compte des critères plus spécifiques à ce type de technologie.

La solution doit proposer des fonctionnalités de base :

- gestion de tous les types de relations (1-1, 1-n, n-n)
- langage de requêtes supportant des fonctions avancées de SQL (jointure, groupage, agrégat, ...)
- support des transactions
- support de l'héritage et du polymorphisme
- support de nombreuses bases de données
- gestion des accès concurrents
- support des différents types de clés et des clés composées
- support des mises à jour en cascade
- support du mapping de type une classe contenant des données de plusieurs tables ou l'inverse
- configuration par fichiers ou annotations
- ...

La prise en compte des performances et des optimisations proposées par la solution est très importante :

- chargement différé (lazy loading) au niveau d'un champ
- gestion de caches au niveau des données et des requêtes
- optimisation des requêtes
- ...

Il est aussi nécessaire de tenir compte des outils proposés par la solution pour faciliter sa mise en oeuvre. Ces outils peuvent par exemple :

- permettre l'automatisation de la génération des classes ou des schémas de la base de données
- faciliter la rédaction et la maintenant des fichiers de configuration
- ...

Certaines fonctionnalités avancées peuvent être utiles voir requises en fonction des besoins :

- mise en oeuvre de POJO
- support du mode déconnecté
- support des procédures stockées
- ...

39.3.2. Les difficultés lors de la mise en place d'un outil de mapping O/R

De nombreuses difficultés peuvent survenir lors de la mise en oeuvre d'un outil de mapping O/R

- Difficultés à mapper le modèle relationnel à cause de la complexité du modèle ou de sa mauvaise conception
- Temps d'apprentissage de l'outil plus ou moins important
- Difficultés pour mettre en oeuvre les transactions
- Parfois des problèmes de performance peuvent nécessiter un paramétrage plus fin notamment en se qui concerne la mise en oeuvre de caches ou du chargement tardif (lazy loading)
- Difficultés pour maintenir les fichiers de configuration généralement au format XML et à les synchroniser avec les évolutions du modèle de données. Des outils existent pour certaines solutions afin de faciliter cette tâche.

39.4. L'architecture et la persistance de données

Dans une architecture en couche, il est important de prévoir une couche dédiée aux accès aux données.

Il est assez fréquent dans cette couche de parler de la notion de CRUD qui représentent un ensemble des 4 opérations de bases réalisable sur une données.

Il est aussi de bon usage de mettre en oeuvre le design pattern DAO (Data Access Object) proposé par Sun.

39.4.1. La couche de persistance

La partie du code responsable de l'accès aux données dans une application multi niveaux doit être encapsulée dans une couche dédiée aux interactions avec la base de données de l'architecture généralement appelée couche de persistance. Celle-ci permet notamment :

- d'ajouter un niveau d'abstraction entre la base de données et l'utilisation qui en est faite.
- de simplifier la couche métier qui utilise les traitements de cette couche
- de masquer les traitements réalisés pour mapper les objets dans la base de données et vice et versa
- de faciliter le remplacement de la base de données utilisée

La couche métier qui va utiliser la couche de persistance reste indépendante du code dédié à l'accès à la base de données. Ainsi la couche métier ne contient aucune requête SQL, ni code de connexion ou d'accès à la base de données. La couche

métier utilise les classes de la couche métier qui encapsulent ces traitements. Ainsi la couche métier manipule uniquement des objets pour les accès à la base de données.

Le choix des API ou des outils dépends du contexte : certaines solutions ne sont utilisables qu'avec la plate-forme Enterprise Edition (exemple : les EJB) ou sont utilisables indifféremment avec les plates-formes Standard et Enterprise Edition.

L'utilisation d'une API standard permet de garantir la pérennité et de choisir l'implémentation à mettre en oeuvre.

Les solutions open source et commerciales ont les avantages et inconvénients inhérents à leur typologie respective.

39.4.2. Les opérations de type CRUD

L'acronyme CRUD (Create, Read, Update and Delete) désigne les quatre opérations réalisables sur des données (création, lecture, mise à jour et suppression).

Exemple : une interface qui propose des opérations de type CRUD pour un objet de type Entite

```
public interface EntiteCrud {
  public Entite obtenir(Integer id);
  public void creer(Entite entite);
  public void modifier(Entite entite);
  public Collection obtenirTous();
  public void supprimer(Entite entite);
}
```

39.4.3. Le modèle de conception DAO (Data Access Object)

DAO est l'acronyme de Data Access Object. C'est un modèle de conception qui propose de découpler l'accès à une source de données.

L'accès aux données dépend fortement de la source de données. Par exemple, l'utilisation d'une base de données est spécifique pour chaque fournisseur. Même si SQL et JDBC assurent une partie de l'indépendance vis-à-vis de la base de données utilisées, certaines contraintes imposent une mise à en oeuvre spécifique de certaines fonctionnalités.

Par exemple, la gestion des champs de type identifiants est proposée selon diverses formes par les bases de données : champ auto-incrémenté, identity, séquence, ...

Le motif de conception DAO proposé dans le blue print de Sun propose de séparer les traitements d'accès physique à une source de données de leur utilisation dans les objets métiers. Cette séparation permet de modifier une source de données sans avoir à modifier les traitements qui l'utilise.

Le DAO peut aussi proposer un mécanisme pour rendre l'accès aux bases de données indépendant de la base de données utilisées et même rendre celle-ci paramétrable.

Les classes métier utilisent le DAO via son interface et sont donc indépendantes de son implémentation. Si cette implémentation change (par exemple un changement de base de données), seul l'implémentation du DAO est modifié mais les classes qui l'utilisent via son interface ne sont pas impactées.

Le DAO définit donc une interface qui va exposer les fonctionnalités utilisables. Ces fonctionnalités doivent être indépendantes de l'implémentation sous jacente. Par exemple, aucune méthode ne doit avoir de requêtes SQL en paramètre. Pour les même raisons, le DAO doit proposer sa propre hiérarchie d'exceptions.

Une implémentation concrète de cette interface doit être proposée. Cette implémentation peut être plus ou moins complexe en fonction de critères de simplicité ou de flexibilité.

Fréquemment les DAO ne mettent pas en oeuvre certaines fonctionnalités comme la mise en oeuvre d'un cache ou la

gestion des accès concurrents.

39.5. Les différentes solutions

Différentes solutions peuvent être utilisées pour la persistance des objets en Java :

- Sérialisation
- JDBC
- Génération automatisée de code source
- SQL/J
- Enrichissement du code source (enhancement)
- Génération de byte code
- framework de mapping O/R (Object Relational Mapping)
- Base de données objet (ODBMS)

La plupart de ces solutions offre de surcroît un choix plus ou moins important d'implémentations.

39.6. Les API standards

Les différentes évolutions de Java ont apportées plusieurs solutions pour assurer la persistance des données vers une base de données.

39.6.1. JDBC

JDBC est l'acronyme de Java DataBase Connectivity. C'est l'API standard pour permettre un accès à une base de données. Son but est de permettre de coder des accès à une base de données en laissant le code le plus indépendant de la base de données utilisée.

C'est une spécification qui définit des interfaces pour se connecter et interagir avec la base de données (exécution de requêtes ou de procédures stockées, parcourt des résultats des requêtes de sélection, ...)

L'implémentation de ces spécifications est fournis par des tiers, et en particulier les fournisseurs de base de données, sous la forme de Driver.

La mise en oeuvre de JDBC est détaillée dans le chapitre JDBC

39.6.2. JDO 1.0

JDO est l'acronyme de Java Data Object : le but de cet API est de rendre transparent la persistance d'un objet. Il repose sur l'enrichissement de byte-code à la compilation.

Cette API a été spécifiée sous la JSR-012

Il existe plusieurs implémentations dont :

- Apache JDO (http://db.apache.org/jdo/)
- JPOX (http://www.jpox.org)
- Xcalia LiDO
- Kodo (http://www.solarmetric.com/) - BEA
- Speedo (http://speedo.objectweb.org/)

La mise en oeuvre de JDO est détaillée dans le chapitre JDO

39.6.3. JD0 2.0

La version 2 de JDO a été diffusée en mars 2006.

Il existe de plusieurs implémentations dont :

- Apache JDO (http://db.apache.org/jdo/)
- JPOX (http://www.jpox.org) : c'est l'implémentation de référence (RI) pour JDO 2.0
- Kodo (http://www.solarmetric.com/) - BEA

39.6.4. EJB 2.0

Les EJB (Enterprise Java Bean) proposent des beans de type Entités pour assurer la persistance des objets.

Les EJB de type Entité peuvent être de deux types :

- CMP (Container Managed Persistance) : la persistance est assuré par le conteneur d'EJB en fonction du paramétrage fourni
- BMP (Bean Managed Persistance) :

Les EJB bénéficient des services proposés par le conteneur cependant cela les rends dépendant de ce conteneur pour l'exécution : ils sont difficilement utilisables en dehors du conteneur (par exemple pour les tester).

Il existe de nombreuses implémentations puisque chaque serveur d'application certifié J2EE doit implémenter les EJB ce qui inclus entre autre JBoss de RedHat, JonAS, Geronimo d'Apache, GlassFish de Sun, Websphere d'IBM, Weblogic de BEA, ...

39.6.5. Java Persistence API et EJB 3.0

JPA (Java Persistence API) est issu des travaux de la JSR-220 concernant la version 3.0 des EJB : elle remplace d'ailleurs les EJB Entités version 2. C'est une synthèse standardisée des meilleurs outils du sujet (Hibernate, Toplink, ...)

L'API repose sur

- l'utilisation d'entités persistantes sous la forme de POJOs
- un gestionnaire de persistance (EntityManager) qui assure la gestion des entités persistantes
- l'utilisation d'annotations
- la configuration via des fichiers xml

JPA peut être utilisé avec Java EE dans un serveur d'application mais aussi avec Java SE (avec quelques fonctionnalités proposées par le conteneur en moins).

JPA est une spécification : il est nécessaire d'utiliser une implémentation pour la mettre en oeuvre. L'implémentation de référence est la partie open source d'Oracle Toplink : Toplink essential. La version 3.2 d'Hibernate implémente aussi JPA.

JPA ne peut être utilisé qu'avec des bases de données relationnelles.

La version 3.0 des EJB utilise JPA pour la persistance des données.

La mise en oeuvre de JPA est détaillée dans le chapitre JPA

39.7. Les frameworks open source

Pour palier à certaines faiblesses des API standards, la communauté open source a développé de nombreux frameworks concernant la persistance de données dont le plus utilisé est Hibernate. Cette section va rapidement présentés quelques uns d'entre eux.

39.7.1. iBatis

Le site officiel du projet est à l'url : http://ibatis.apache.org/

39.7.2. Hibernate

Hibernate est le framework open source de mapping O/R le plus populaire. Cette popularité est liée à la richesse des fonctionnalités proposées et à ses performances.

Hibernate propose son propre langage d'interrogation HQL et a largement inspiré les concepteurs de l'API JPA. Hibernate est un projet open source de mapping O/R qui fait référence en la matière car il possède plusieurs avantages :

- manipulation de données d'une base de données relationnelles à partir d'objets Java
- facile à mettre en oeuvre, efficace et fiable
- open source

Le site officiel du projet est à l'url : http://www.hibernate.org/

L'utilisation d'Hibernate est détaillée dans le chapitre Hibernate

39.7.3. Castor

Castor permet de mapper des données relationnelles ou XML avec des objets Java.

Castor propose une solution riche mais qui n'implémente aucun standard.

Le site officiel du projet est à l'url : http://www.castor.org/

39.7.4. Apache Torque

Torque est un framework initialement développé pour le projet Jakarta Turbine : il est développé depuis sous la forme d'un projet autonome.

Torque se compose d'un générateur qui va générer automatiquement les classes requises pour accéder à la base de données et d'un environnement d'exécution qui va permet la mise en oeuvre des classes générées.

Torque utilise un fichier XML contenant une description de la base de données pour générer des classes permettant des opérations sur la base de données grâce à des outils dédiés. Ces classes reposent sur un environnement d'exécution (runtime) fourni par Torque.

Le site officiel du projet est à l'url http://db.apache.org/torque/

39.7.5. TopLink

TopLink a été racheté par Oracle et intégré dans ses solutions J2EE.

Le site officiel du produit est à l'url : http://www.oracle.com/technology/products/ias/toplink/index.html

La partie qui compose la base de TopLink est en open source.

39.7.6. Apache OJB

Le site officiel du projet est à l'url http://db.apache.org/ojb/

39.7.7. Apache Cayenne

Le site officiel du projet est à l'url http://cayenne.apache.org/

Cayenne est distribué avec un outil de modélisation nommé CayenneModeler.

39.8. L'utilisation de procédures stockées

L'utilisation de procédures stockées peut apporter des améliorations notamment en termes de performance de certains traitements.

Cependant, les procédures stockées possèdent plusieurs inconvénients :

- peu portable
- peu flexible
- maintenance généralement difficile liée au manque d'outillage
- ...

40. JDBC (Java DataBase Connectivity)

JDBC est l'acronyme de Java DataBase Connectivity et désigne une API définie par Sun pour permettre un accès aux bases de données avec Java.

Ce chapitre présente dans plusieurs sections l'utilisation de cette API :

- Les outils nécessaires pour utiliser JDBC
- Les types de pilotes JDBC
- L'enregistrement d'une base de données dans ODBC sous Windows 9x ou XP
- La présentation des classes de l'API JDBC
- La connexion à une base de données
- L'accès à la base de données
- L'obtention d'informations sur la base de données
- L'utilisation d'un objet PreparedStatement
- L'utilisation des transactions
- Les procédures stockées
- Le traitement des erreurs JDBC
- JDBC 2.0
- JDBC 3.0
- MySQL et Java
- L'amélioration des performances avec JDBC
- Les ressources relatives à JDBC

40.1. Les outils nécessaires pour utiliser JDBC

Les classes de JDBC version 1.0 sont regroupées dans le package java.sql et sont incluses dans le JDK à partir de sa version 1.1. La version 2.0 de cette API est incluse dans la version 1.2 du JDK.

Pour pouvoir utiliser JDBC, il faut un pilote qui est spécifique à la base de données à laquelle on veut accéder. Avec le JDK, Sun fournit un pilote qui permet l'accès aux bases de données via ODBC.

Ce pilote permet de réaliser l'indépendance de JDBC vis à vis des bases de données.

Pour utiliser le pont JDBC-ODBC sous Window 9x, il faut utiliser ODBC en version 32 bits.

40.2. Les types de pilotes JDBC

Il existe quatre types de pilote JDBC :

1. Type 1 (JDBC-ODBC bridge) : le pont JDBC-ODBC qui s'utilise avec ODBC et un pilote ODBC spécifique pour la base à accéder. Cette solution fonctionne très bien sous Windows. C'est une solution pour des développements avec exécution sous Windows d'une application locale qui a le mérite d'être universel car il existe des pilotes ODBC pour la quasi totalité des bases de données. Cette solution "simple" pour le développement possède plusieurs inconvénients :

- ♦ la multiplication du nombre de couches rend complexe l'architecture (bien que transparent pour le développeur) et détériore un peu les performances
- ♦ lors du déploiement, ODBC et son pilote doivent être installé sur tous les postes ou l'application va fonctionner.
- ♦ la partie native (ODBC et son pilote) rend l'application moins portable et dépendante d'une plateforme.

2. Type 2 : un driver écrit en java qui appelle l'API native de la base de données

   Ce type de driver convertit les ordres JDBC pour appeler directement les API de la base de données via un pilote natif sur le client. Ce type de driver nécessite aussi l'utilisation de code natif sur le client.

3. Type 3 : un driver écrit en Java utilisant un middleware

   Ce type de driver utilise un protocole réseau propriétaire spécifique à une base de données. Un serveur dédié reçoit les messages par ce protocole et dialogue directement avec la base de données. Ce type de driver peut être facilement utilisé par une applet mais dans ce cas le serveur intermédiaire doit obligatoirement être installé sur la machine contenant le serveur web.

4. Type 4 : un driver Java utilisant le protocole natif de la base de données

   Ce type de driver, écrit en java, appelle directement le SGBD par le réseau. Ils sont fournis par l'éditeur de la base de données.

Les drivers se présentent souvent sous forme de fichiers jar dont le chemin doit être ajouté au classpath pour permettre au programme de l'utiliser.

Sun maintient une liste des drivers jdbc à l'url : http://www.javasoft.com/products/jdbc/drivers.html

40.3. L'enregistrement d'une base de données dans ODBC sous Windows 9x ou XP

Pour utiliser un pilote de type 1 (pont ODBC-JDBC) sous Windows 9x, il est nécessaire d'enregistrer la base de données dans ODBC avant de pouvoir l'utiliser.

Attention : ODBC n'est pas fourni en standard avec Windows 9x.

Pour enregistrer une nouvelle base de données, il faut utiliser l'administrateur de source de données ODBC.

Pour lancer cette application sous Windows 9x, il faut doubler cliquer sur l'icône "ODBC 32bits" dans le panneau de configuration.

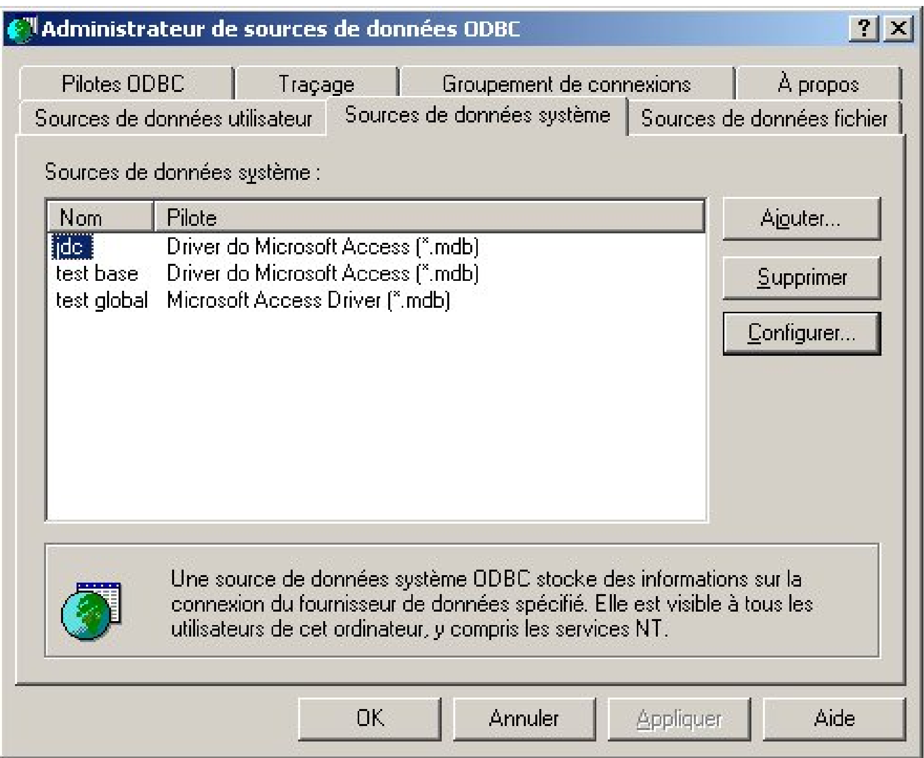
Sous Windows XP, il faut double cliquer sur l'icône "Source de données (ODBC)" dans le répertoire "Outils d'administration" du panneau de configuration.

L'outil se compose de plusieurs onglets.

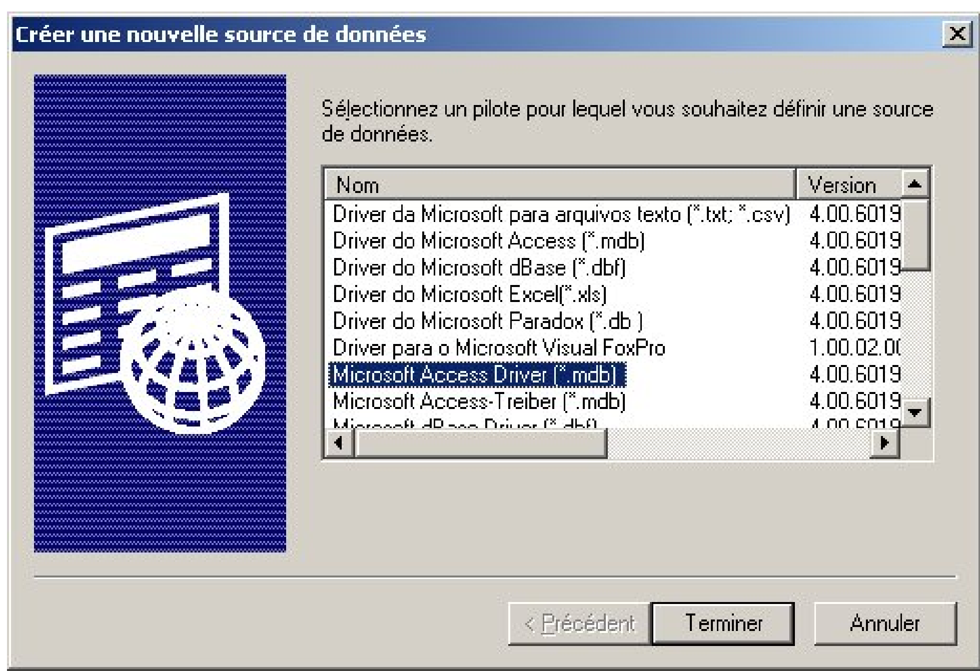
L'onglet "Pilote ODBC" liste l'ensemble des pilotes qui sont installés sur la machine.

L'onglet "Source de données utilisateur" liste l'ensemble des sources de données pour l'utilisateur couramment connecté sous Windows.

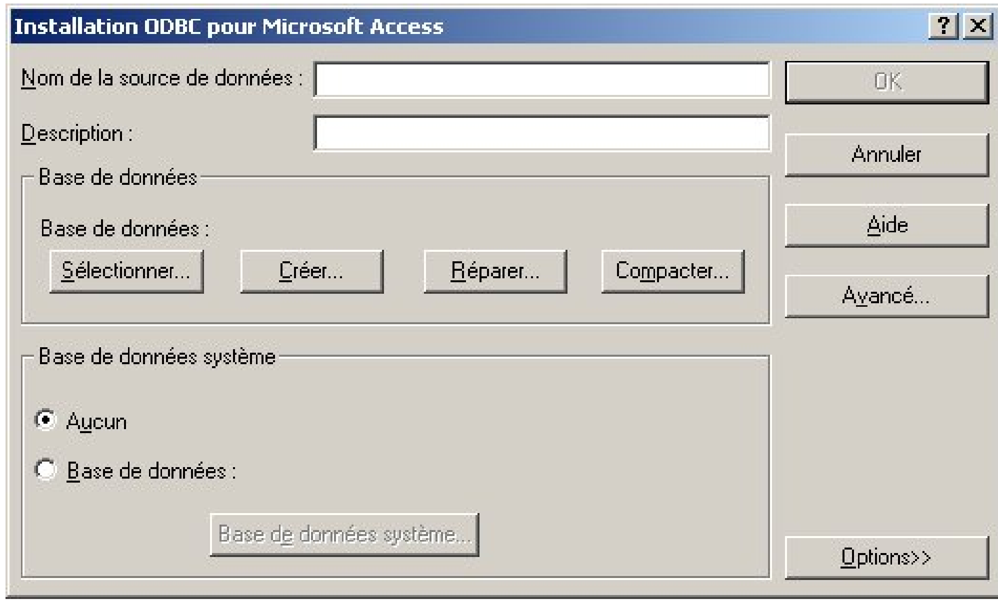
L'onglet "Source de données système" liste l'ensemble des sources de données accessibles par tous les utilisateurs.



Le plus simple est de créer une telle source de données en cliquant sur le bouton "Ajouter". Une boite de dialogue permet de sélectionner le pilote qui sera utilisé par la source de données.



Il suffit de sélectionner le pilote et de cliquer sur "Terminer". Dans l'exemple ci dessous, le pilote sélectionné concerne une base Microsoft Access.



Il suffit de saisir les informations nécessaires notamment le nom de la source de données et de sélectionner la base. Un clic sur le bouton "Ok" crée la source de données qui pourra alors être utilisée.

40.4. La présentation des classes de l'API JDBC

Toutes les classes de JDBC sont dans le package java.sql. Il faut donc l'importer dans tous les programmes devant utiliser JDBC.

Exemple :

```
import java.sql.*;
```

Il y a 4 classes importantes : DriverManager, Connection, Statement (et PreparedStatement), et ResultSet, chacune correspondant à une étape de l'accès aux données :

Classe	Rôle
DriverManager	charge et configure le driver de la base de données.
Connection	réalise la connexion et l'authentification à la base de données.
Statement (et PreparedStatement)	contient la requête SQL et la transmet à la base de données.
ResultSet	permet de parcourir les informations retournées par la base de données dans le cas d'une sélection de données

Chacune de ces classes dépend de l'instanciation d'un objet de la précédente classe.

40.5. La connexion à une base de données

40.5.1. Le chargement du pilote

Pour se connecter à une base de données via ODBC, il faut tout d'abord charger le pilote JDBC-ODBC qui fait le lien entre les deux.

Exemple (code Java 1.1) :

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

Pour se connecter à une base en utilisant un driver spécifique, la documentation du driver fournit le nom de la classe à utiliser. Par exemple, si le nom de la classe est jdbc.DriverXXX, le chargement du driver se fera avec le code suivant :

Class.forName("jdbc.DriverXXX");

Exemple : Chargement du pilote pour un base PostgreSQL sous Linux

```
Class.forName("postgresql.Driver");
```

Il n'est pas nécessaire de créer une instance de cette classe et de l'enregistrer avec le DriverManager car l'appel à Class.forName le fait automatiquement : ce traitement charge le pilote et créer une instance de cette classe.

La méthode static forName() de la classe Class peut lever l'exception java.lang.ClassNotFoundException.

40.5.2. L'établissement de la connexion

Pour se connecter à une base de données, il faut instancier un objet de la classe Connection en lui précisant sous forme d'URL la base à accéder.

Exemple (code Java 1.1) : Etablir une connexion sur la base testDB via ODBC

```
String DBurl = "jdbc:odbc:testDB";

con = DriverManager.getConnection(DBurl);
```

La syntaxe URL peut varier d'un type de base de données à l'autre mais elle est toujours de la forme : protocole:sous_protocole:nom

"jbdc" désigne le protocole est vaut toujours "jdbc". "odbc" désigne le sous protocole qui définit le mécanisme de connexion pour un type de bases de données.

Le nom de la base de données doit être celui saisi dans le nom de la source sous ODBC.

La méthode getConnection() peut lever une exception de la classe java.sql.SQLException.

Le code suivant décrit la création d'une connexion avec un user et un mot de passe :

Exemple (code Java 1.1) :

```
Connection con = DriverManager.getConnection(url, "myLogin", "myPassword");
```

A la place de " myLogin " ; il faut mettre le nom du user qui se connecte à la base et mettre son mot de passe à la place de "myPassword "

Exemple (code Java 1.1) :

```
String url = "jdbc:odbc:factures";

Connection con = DriverManager.getConnection(url, "toto", "passwd");
```

La documentation d'un autre driver indiquera le sous protocole à utiliser (le protocole à mettre derrière jdbc dans l'URL).

Exemple : Connection à la base PostgreSQL nommée test avec le user jumbo et le mot de passe 12345 sur la machine locale

```
Connection con=DriverManager.getConnection("jdbc:postgresql://localhost/test","jumbo","12345");
```

40.6. L'accès à la base de données

Une fois la connexion établie, il est possible d'exécuter des ordres SQL. Les objets qui peuvent être utilisés pour obtenir des informations sur la base de données sont :

Classe	Rôle
DatabaseMetaData	informations à propos de la base de données : nom des tables, index, version ...
ResultSet	résultat d'une requête et information sur une table. L'accès se fait enregistrement par enregistrement.
ResultSetMetaData	informations sur les colonnes (nom et type) d'un ResultSet

40.6.1. L'exécution de requêtes SQL

Les requêtes d'interrogation SQL sont exécutées avec les méthodes d'un objet Statement que l'on obtient à partir d'un objet Connection

Exemple (code Java 1.1) :

```
ResultSet résultats = null;
String requete = "SELECT * FROM client";

try {
   Statement stmt = con.createStatement();
   résultats = stmt.executeQuery(requete);
} catch (SQLException e) {
   //traitement de l'exception
}
```

Un objet de la classe Statement permet d'envoyer des requêtes SQL à la base. La création d'un objet Statement s'effectue à partir d'une instance de la classe Connection :

Exemple (code Java 1.1) :

```
Statement stmt = con.createStatement();
```

Pour une requête de type interrogation (SELECT), la méthode à utiliser de la classe Statement est executeQuery(). Pour des traitements de mise à jour, il faut utiliser la méthode executeUpdate(). Lors de l'appel à la méthode d'exécution, il est nécessaire de lui fournir en paramètre la requête SQL sous forme de chaine.

Le résultat d'une requête d'interrogation est renvoyé dans un objet de la classe ResultSet par la méthode executeQuery().

Exemple (code Java 1.1) :

```
ResultSet rs = stmt.executeQuery("SELECT * FROM employe");
```

La méthode executeUpdate() retourne le nombre d'enregistrements qui ont été mis à jour

Exemple (code Java 1.1) :

```
...

//insertion d'un enregistrement dans la table client

requete = "INSERT INTO client VALUES (3,'client 3','prenom 3')";
try {
   Statement stmt = con.createStatement();
   int nbMaj = stmt.executeUpdate(requete);
   affiche("nb mise a jour = "+nbMaj);
} catch (SQLException e) {
   e.printStackTrace();
}

...
```

Lorsque la méthode executeUpdate() est utilisée pour exécuter un traitement de type DDL (Data Definition Langage : définition de données) comme la création d'un table, elle retourne 0. Si la méthode retourne 0, cela peut signifier deux choses : le traitement de mise à jour n'a affecté aucun enregistrement ou le traitement concernait un traitement de type DDL.

Si l'on utilise executeQuery() pour exécuter une requête SQL ne contenant pas d'ordre SELECT, alors une exception de type SQLException est levée.

Exemple (code Java 1.1) :

```
...

requete = "INSERT INTO client VALUES (4,'client 4','prenom 4')";
try {
   Statement stmt = con.createStatement();
   ResultSet résultats = stmt.executeQuery(requete);
} catch (SQLException e) {
   e.printStackTrace();
}

...
```

Résultat :

```
java.sql.SQLException: No ResultSet was produced
java.lang.Throwable(java.lang.String)
java.lang.Exception(java.lang.String)
java.sql.SQLException(java.lang.String)
java.sql.ResultSet sun.jdbc.odbc.JdbcOdbcStatement.executeQuery(java.lang.String)
void testjdbc.TestJDBC1.main(java.lang.String [])
```

Attention : dans ce cas la requête est quand même effectuée. Dans l'exemple, un nouvel enregistrement est créé dans la table.

Il n'est pas nécessaire de définir un objet Statement pour chaque ordre SQL : il est possible d'un définir un et de le réutiliser

40.6.2. La classe ResultSet

C'est une classe qui représente une abstraction d'une table qui se compose de plusieurs enregistrements constitués de colonnes qui contiennent les données.

Les principales méthodes pour obtenir des données sont :

Méthode	Rôle
getInt(int)	retourne le contenu de la colonne dont le numéro est passé en paramètre sous forme d'entier.
getInt(String)	retourne le contenu de la colonne dont le nom est passé en paramètre sous forme d'entier.
getFloat(int)	retourne le contenu de la colonne dont le numéro est passé en paramètre sous forme de nombre flottant.
getFloat(String)	
getDate(int)	retourne le contenu de la colonne dont le numéro est passé en paramètre sous forme de date.
getDate(String)	
next()	se déplace sur le prochain enregistrement : retourne false si la fin est atteinte
Close()	ferme le ResultSet
getMetaData()	retourne un objet ResultSetMetaData associé au ResultSet.

La méthode getMetaData() retourne un objet de la classe ResultSetMetaData qui permet d'obtenir des informations sur le résultat de la requête. Ainsi, le nombre de colonne peut être obtenu grâce à la méthode getColumnCount() de cet objet.

Exemple :

```
ResultSetMetaData rsmd;
rsmd = results.getMetaData();
nbCols = rsmd.getColumnCount();
```

La méthode next() déplace le curseur sur le prochain enregistrement. Le curseur pointe initialement juste avant le premier enregistrement : il est nécessaire de faire un premier appel à la méthode next() pour se placer sur le premier enregistrement.

Des appels successifs à next permettent de parcourir l'ensemble des enregistrements.

Elle retourne false lorsqu'il n'y a plus d'enregistrement. Il faut toujours protéger le parcours d'une table dans un bloc de capture d'exception

Exemple (code Java 1.1) :

```
//parcours des données retournées

try {
   ResultSetMetaData rsmd = résultats.getMetaData();
   int nbCols = rsmd.getColumnCount();
   boolean encore = résultats.next();
   while (encore) {
      for (int i = 1; i <= nbCols; i++)
         System.out.print(résultats.getString(i) + " ");
      System.out.println();
      encore = résultats.next();
   }
   résultats.close();
```

```
} catch (SQLException e) {
   //traitement de l'exception
}
```

Les méthodes getXXX() permettent d'extraire les données selon leur type spécifiée par XXX tel que getString(), getDouble(), getInteger(), etc ... . Il existe deux formes de ces méthodes : indiquer le numéro la colonne en paramètre (en commençant par 1) ou indiquer le nom de la colonne en paramètre. La première méthode est plus efficace mais peut générer plus d'erreurs à l'exécution notamment si la structure de la table évolue.

Attention : il est important de noter que ce numéro de colonne fourni en paramètre fait référence au numéro de colonne de l'objet resultSet (celui correspondant dans l'ordre SELECT)et non au numéro de colonne de la table.

La méthode getString() permet d'obtenir la valeur d'un champ de n'importe quel type.

40.6.3. Un exemple complet de mise à jour et de sélection sur une table

Exemple (code Java 1.1) :

```
import java.sql.*;

public class TestJDBC1 {

   private static void affiche(String message) {
      System.out.println(message);
   }

   private static void arret(String message) {
      System.err.println(message);
      System.exit(99);
   }

   public static void main(java.lang.String[] args) {
      Connection con = null;
      ResultSet résultats = null;
      String requete = "";

      // chargement du pilote
      try {
         Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      } catch (ClassNotFoundException e) {
         arret("Impossible de charger le pilote jdbc:odbc");
      }

      //connection a la base de données

      affiche("connexion a la base de données");
      try {

         String DBurl = "jdbc:odbc:testDB";
         con = DriverManager.getConnection(DBurl);
      } catch (SQLException e) {
         arret("Connection à la base de données impossible");
      }

      //insertion d'un enregistrement dans la table client
      affiche("creation enregistrement");

      requete = "INSERT INTO client VALUES (3,'client 3','client 4')";
      try {
         Statement stmt = con.createStatement();
         int nbMaj = stmt.executeUpdate(requete);
         affiche("nb mise a jour = "+nbMaj);
      } catch (SQLException e) {
          e.printStackTrace();
```

```
      }

      //creation et execution de la requete
      affiche("creation et execution de la requête");
      requete = "SELECT * FROM client";

      try {
         Statement stmt = con.createStatement();
         résultats = stmt.executeQuery(requete);
      } catch (SQLException e) {
         arret("Anomalie lors de l'execution de la requête");
      }

      //parcours des données retournées
      affiche("parcours des données retournées");
      try {
         ResultSetMetaData rsmd = résultats.getMetaData();
         int nbCols = rsmd.getColumnCount();
         boolean encore = résultats.next();

         while (encore) {

            for (int i = 1; i <= nbCols; i++)
               System.out.print(résultats.getString(i) + " ");
            System.out.println();
            encore = résultats.next();
         }

         résultats.close();
      } catch (SQLException e) {
         arret(e.getMessage());
      }

      affiche("fin du programme");
      System.exit(0);
   }
}
```

Résultat :

```
connexion a la base de données
creation enregistrement
nb mise a jour = 1
creation et execution de la requête
parcours des données retournées
1.0 client 1 prenom 1
2.0 client 2 prenom 2
3.0 client 3 client 4
fin du programme
```

40.7. L'obtention d'informations sur la base de données

40.7.1. La classe ResultSetMetaData

La méthode getMetaData() d'un objet ResultSet retourne un objet de typeResultSetMetaData. Cet objet permet de connaître le nombre, le nom et le type des colonnes.

Méthode	Rôle
int getColumnCount()	retourne le nombre de colonnes du ResultSet
String getColumnName(int)	retourne le nom de la colonne dont le numéro est donné
String getColumnLabel(int)	retourne le libellé de la colonne donnée

boolean isCurrency(int)	retourne true si la colonne contient un nombre au format monétaire
boolean isReadOnly(int)	retourne true si la colonne est en lecture seule
boolean isAutoIncrement(int)	retourne true si la colonne est auto incrémentée
int getColumnType(int)	retourne le type de données SQL de la colonne

40.7.2. La classe DatabaseMetaData

Un objet de la classe DatabaseMetaData permet d'obtenir des informations sur la base de données dans son ensemble : nom des tables, nom des colonnes dans une table, méthodes SQL supportées

Méthode	Rôle
ResultSet getCatalogs()	retourne la liste du catalogue d'informations (Avec le pont JDBC-ODBC, on obtient la liste des bases de données enregistrées dans ODBC).
ResultSet getTables(catalog, schema, tableNames, columnNames)	retourne une description de toutes les tables correspondant au TableNames donné et à toutes les colonnes correspondantes à columnNames.
ResultSet getColumns(catalog, schema, tableNames, columnNames)	retourne une description de toutes les colonnes correspondantes au TableNames donné et à toutes les colonnes correspondantes à columnNames.
String getURL()	retourne l'URL de la base à laquelle on est connecté
String getDriverName()	retourne le nom du driver utilisé

La méthode getTables() de l'objet DataBaseMetaData demande quatre arguments :

getTables(catalog, schema, tablemask, types[]);

- catalog : le nom du catalogue dans lequel les tables doivent être recherchées. Pour une base de données JDBC-ODBC, il peut être mis à null.
- schema : le schéma de la base de données à inclure pour les bases les supportant. Il est en principe mis à null
- tablemask : un masque décrivant les noms des tables à retrouver. Pour les retrouver toutes, il faut l'initialiser avec le caractères '%'
- types[] : tableau de chaines décrivant le type de tables à retrouver. La valeur null permet de retrouver toutes les tables.

Exemple (code Java 1.1) :

```
con = DriverManager.getConnection(url);
dma =con.getMetaData();
String[] types = new String[1];
types[0] = "TABLES"; //set table type mask

results = dma.getTables(null, null, "%", types);

boolean more = results.next();
while (more) {
   for (i = 1; i <= numCols; i++)
      System.out.print(results.getString(i)+" ");
   System.out.println();
   more = results.next();
}
```

40.8. L'utilisation d'un objet PreparedStatement

L'interface PreparedStatement définit les méthodes pour un objet qui va encapsuler une requête précompilée. Ce type de requête est particulièrement adapté pour une exécution répétée d'une même requête avec des paramètres différents.

Cette interface hérite de l'interface Statement.

Lors de l'utilisation d'un objet de type PreparedStatement, la requête est envoyée au moteur de la base de données pour que celui ci prépare son exécution.

Un objet qui implémente l'interface PreparedStatement est obtenu en utilisant la méthode preparedStatement() d'un objet de type Connection. Cette méthode attend en paramètre une chaîne de caractères contenant la requête SQL. Dans cette chaine, chaque paramètre est représenté par un caractère ?.

Un ensemble de méthode setXXX() (ou XXX représente un type primitif ou certains objets tel que String, Date, Object, ...) permet de fournir les valeurs de chaque paramètre défini dans la requête. Le premier paramètre de ces méthodes précise le numéro du paramètre dont la méthode va fournir la valeur. Le second paramètre précise cette valeur.

Exemple (code Java 1.1) :

```
package com.jmd.test.dej;

import java.sql.*;

public class TestJDBC2 {

  private static void affiche(String message) {
    System.out.println(message);
  }

  private static void arret(String message) {
    System.err.println(message);
    System.exit(99);
  }

  public static void main(java.lang.String[] args) {
    Connection con = null;
    ResultSet resultats = null;
    String requete = "";

    try {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    } catch (ClassNotFoundException e) {
      arret("Impossible de charger le pilote jdbc:odbc");
    }

    affiche("connexion a la base de données");
    try {

      String DBurl = "jdbc:odbc:testDB";
      con = DriverManager.getConnection(DBurl);
      PreparedStatement recherchePersonne =
        con.prepareStatement("SELECT * FROM personnes WHERE nom_personne = ?");

      recherchePersonne.setString(1, "nom3");

      resultats = recherchePersonne.executeQuery();

      affiche("parcours des données retournées");

      boolean encore = resultats.next();

      while (encore) {
        System.out.print(resultats.getInt(1) + " :  "+resultats.getString(2)+" "+
          resultats.getString(3)+"("+resultats.getDate(4)+")");
        System.out.println();
        encore = resultats.next();
      }

      resultats.close();
```

```
    } catch (SQLException e) {
      arret(e.getMessage());
    }

    affiche("fin du programme");
    System.exit(0);
  }
}
```

Pour exécuter la requête, l'interface PreparedStatement propose deux méthodes :

- executeQuery() : cette méthode permet d'exécuter une requête de type interrogation et renvoie un objet de type ResultSet qui contient les données issues de l'exécution de la requête
- executeUpdate() : cette méthode permet d'exécuter une requête de type mise à jour et renvoie un entier qui contient le nombre d'occurrences impactées par la mise à jour

40.9. L'utilisation des transactions

Une transaction permet de ne valider un ensemble de traitements sur la base de données que s'ils se sont tous effectués correctement.

Par exemple, une opération bancaire de transfert de fond d'un compte vers un autre oblige à la réalisation de l'opération de débit sur un compte et de l'opération de crédit sur l'autre compte. La réalisation d'une seule de ces opérations laisserait les données de la base dans un état inconsistant.

Une transaction est un mécanisme qui permet donc de s'assurer que toutes les opérations qui la compose seront réellement effectuées.

Une transaction est gérée à partir de l'objet Connection. Par défaut, une connexion est en mode auto-commit. Dans ce mode, chaque opération est validée unitairement pour former la transaction.

Pour pouvoir rassembler plusieurs traitements dans une transaction, il faut tout d'abord désactiver le mode auto-commit. La classe Connection possède la méthode setAutoCommit() qui attend un booléen qui précise le mode de fonctionnement.

Exemple :

connection.setAutoCommit(false);

Une fois le mode auto-commit désactivé, un appel à la méthode commit() de la classe Connection permet de valider la transaction courante. L'appel à cette méthode valide la transaction courante et créé implicitement une nouvelle transaction.

Si une anomalie intervient durant la transaction, il est possible de faire un retour en arrière pour revenir à la situation de la base de données au début de la transaction en appelant la méthode rollback() de la classe Connection.

40.10. Les procédures stockées

L'interface CallableStatement définit les méthodes pour un objet qui va permettre d'appeler une procédure stockée.

Cette interface hérite de l'interface PreparedStatement.

Un objet qui implémente l'interface CallableStatement est obtenu en utilisant la méthode prepareCall() d'un objet de type Connection. Cette méthode attend en paramètre une chaîne de caractères contenant la chaîne d'appel de la procédure stockée.

L'appel d'une procédure étant particulier à chaque base de données supportant une telle fonctionnalité, JDBC propose une syntaxe unifiée qui sera transcrite par le pilote en un appel natif à la base de données. Cette syntaxe peut prendre plusieurs formes :

- {call nom_procedure_stockees} : cette forme la plus simple permet l'appel d'une procédure stockée sans paramètre ni valeur de retour
- {call nom_procedure_stockees(?, ?, ...)} : cette forme permet l'appel d'une procédure stockée avec des paramètres
- {? = call nom_procedure_stockees(?, ?, ...)} : cette forme permet l'appel d'une procédure stockée avec des paramètres et une valeur de retour

Un ensemble de méthode setXXX() (ou XXX représente un type primitif ou certains objets tel que String, Date, Object, ...) permet de fournir les valeurs de chaque paramètre défini dans la requête. Le premier paramètre de ces méthodes précise le numéro du paramètre dont la méthode va fournir la valeur. Le second paramètre précise cette valeur.

Un ensemble de méthode getXXX() (ou XXX représente un type primitif ou certains objets tel que String, Date, Object, ...) permet d'obtenir la valeur du paramètre de retour en fournissant la valeur 0 comme index de départ et un autre index pour les paramètres définis en entrée/sortie dans la procédure stockée.

Pour exécuter la requête, l'interface PreparedStatement propose deux méthodes :

- executeQuery() : cette méthode permet d'exécuter une requête de type interrogation et renvoie un objet de type ResultSet qui contient les données issues de l'exécution de la requête
- executeUpdate() : cette méthode permet d'exécuter une requête de type mise à jour et renvoie un entier qui contient le nombre d'occurrences impactées par la mise à jour

40.11. Le traitement des erreurs JDBC

JDBC permet de connaitre les avertissements et les exceptions générées par la base de données lors de l'exécution de requête.

La classe SQLException représente les erreurs émises par la base de données. Elle contient trois attributs qui permettent de préciser l'erreur :

- message : contient une description de l'erreur
- SQLState : code défini par les normes X/Open et SQL99
- ErrorCode : le code d'erreur du fournisseur du pilote

La classe SQLException possède une méthode getNextException() qui permet d'obtenir les autres exceptions levées durant la requête. La méthode renvoie null une fois la dernière exception renvoyée.

Exemple (code Java 1.1) :

```
package com.jmd.test.dej;

import java.sql.*;

public class TestJDBC3 {

  private static void affiche(String message) {
    System.out.println(message);
  }

  private static void arret(String message) {
    System.err.println(message);
    System.exit(99);
  }

  public static void main(java.lang.String[] args) {
    Connection con = null;
```

```
    ResultSet resultats = null;
    String requete = "";

    try {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    } catch (ClassNotFoundException e) {
      arret("Impossible de charger le pilote jdbc:odbc");
    }

    affiche("connexion a la base de données");
    try {

      String DBurl = "jdbc:odbc:testDB";
      con = DriverManager.getConnection(DBurl);

      requete = "SELECT * FROM tableinexistante";

      Statement stmt = con.createStatement();
      resultats = stmt.executeQuery(requete);

      affiche("parcours des données retournées");

      boolean encore = resultats.next();

      while (encore) {
        System.out.print(resultats.getInt(1) + " :  " + resultats.getString(2) +
          " " + resultats.getString(3) + "(" + resultats.getDate(4) + ")");
        System.out.println();
        encore = resultats.next();
      }

      resultats.close();
    } catch (SQLException e) {
      System.out.println("SQLException");
      do {
        System.out.println("SQLState : " + e.getSQLState());
        System.out.println("Description :  " + e.getMessage());
        System.out.println("code erreur :   " + e.getErrorCode());
        System.out.println("");
        e = e.getNextException();
      } while (e != null);
      arret("");
    } catch (Exception e) {
      e.printStackTrace();
    }

    affiche("fin du programme");
    System.exit(0);
  }
}
```

40.12. JDBC 2.0

La version 2.0 de l'API JDBC a été intégrée au JDK 1.2. Cette nouvelle version apporte plusieurs fonctionnalités très intéressantes dont les principales sont :

- support du parcours dans les deux sens des résultats
- support de la mise à jour des résultats
- possibilité de faire des mises à jour de masse (Batch Updates)
- prise en compte des champs définis par SQL-3 dont BLOB et CLOB

L'API JDBC 2.0 est séparée en deux parties :

- la partie principale (core API) contient les classes et interfaces nécessaires à l'utilisation de bases de données : elles sont regroupées dans le package java.sql

* la seconde partie est une extension utilisée dans J2EE qui permet de gérer les transactions distribuées, les pools de connexions, la connexion avec un objet DataSource ... Les classes et interfaces sont regroupées dans le package javax.sql

40.12.1. Les fonctionnalités de l'objet ResultSet

Les possibilités de l'objet ResultSet dans la version 1.0 de JDBC sont très limitées : parcours séquentiel de chaque occurrence de la table retournée.

La version 2.0 apporte de nombreuses améliorations à cet objet : le parcours des occurrences dans les deux sens et la possibilité de faire des mises à jour sur une occurrence.

Concernant le parcours, il est possible de préciser trois modes de fonctionnement :

* forward-only : parcours séquentiel de chaque occurrence (java.sql.ResultSet.TYPE_FORWARD_ONLY)
* scroll-insensitive : les occurrences ne reflètent pas les mises à jour qui peuvent intervenir durant le parcours (java.sql.ResultSet.TYPE_SCROLL_INSENSITIVE)
* scroll-sensitive : les occurrences reflètent les mises à jour qui peuvent intervenir durant le parcours (java.sql.ResultSet.TYPE_SCROLL_SENSITIVE)

Il est aussi possible de préciser si le ResultSet peut être mise à jour ou non :

* java.sql.ResultSet.CONCUR_READ_ONLY : lecture seule
* java.sql.resultSet.CONCUR_UPDATABLE : mise à jour possible

C'est à la création d'un objet de type Statement qu'il faut préciser ces deux modes. Si ces deux modes ne sont pas précisés, ce sont les caractéristiques de la version 1.0 de JDBC qui sont utilisées (TYPE_FORWARD_ONLY et CONCUR_READ_ONLY).

Exemple (code jdbc 2.0) :

```
Statement statement = connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
  ResultSet.CONCUR_READ_ONLY);
ResultSet resultSet = statement.executeQuery("SELECT nom, prenom FROM employes");
```

Le support de ces fonctionnalités est optionnel pour un pilote. L'objet DatabaseMetadata possède la méthode supportsResultSetType() qui attend en paramètre une constante qui représente une caractéristique : la méthode renvoie un booléen qui indique si la caractéristique est supportée ou non.

A la création du ResultSet, le curseur est positionné avant la première occurrence à traiter. Pour se déplacer dans l'ensemble des occurrences, il y a toujours la méthode next() pour se déplacer sur le suivant mais aussi plusieurs autres méthodes pour permettre le parcours des occurrences en fonctions du mode utilisé dont les principales sont :

Méthode	Rôle
boolean isBeforeFirst()	renvoie un booléen qui indique si la position courante du curseur se trouve avant la première ligne
boolean isAfterLast()	renvoie un booléen qui indique si la position courante du curseur se trouve après la dernière ligne
boolean isFirst()	renvoie un booléen qui indique si le curseur est positionné sur la première ligne
boolean isLast()	renvoie un booléen qui indique si le curseur est positionné sur la dernière ligne

boolean first()	déplace le curseur sur la première ligne
boolean last()	déplace le curseur sur la dernière ligne
boolean absolute()	déplace le curseur sur la ligne dont le numéro est fournie en paramètre à partir du début si il est positif et à partir de la fin si il est négatif. 1 déplace sur la première ligne, -1 sur la dernière, -2 sur l'avant dernière ...
boolean relative(int)	déplace le curseur du nombre de lignes fourni en paramètre par rapport à la position courante du curseur. Le paramètre doit être négatif pour se déplacer vers le début et positif pur se déplacer vers la fin. Avant l'appel de cette méthode, il faut obligatoirement que le curseur soit positionné sur une ligne.
boolean previous()	déplace le curseur sur la ligne précédente. Le boolen indique si la première occurrence est dépassée.
void afterLast()	déplace le curseur après la dernière ligne
void beforeFirst()	déplace le curseur avant la première ligne
int getRow()	renvoie le numéro de la ligne courante

Exemple (code jdbc 2.0) :

```
Statement statement = connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
  ResultSet.CONCUR_READ_ONLY);
ResultSet resultSet = statement.executeQuery(
  "SELECT nom, prenom FROM employes ORDER BY nom");
resultSet.afterLast();
while (resultSet.previous()) {
  System.out.println(resultSet.getString("nom")+
    " "+resultSet.getString("prenom"));
}
```

Durant le parcours d'un ResultSet, il est possible d'effectuer des mises à jour sur la ligne courante du curseur. Pour cela, il faut déclarer l'objet ResultSet comme acceptant les mises à jour. Avec les versions précédentes de JDBC, il fallait utiliser la méthode executeUpdate() avec une requête SQL.

Maintenant pour réaliser ces mises à jour, JDBC 2.0 propose de les réaliser via des appels de méthodes plutôt que d'utiliser des requêtes SQL.

Méthode	Rôle
updateXXX(String, XXX)	permet de mettre à jour la colonne dont le nom est fourni en paramètre. Le type Java de cette colonne est XXX
updateXXX(int, XXX)	permet de mettre à jour la colonne dont l'index est fourni en paramètre. Le type Java de cette colonne est XXX
updateRow()	permet d'actualiser les modifications réalisées avec des appels à updateXXX()
boolean rowsUpdated()	indique si la ligne courante a été modifiée
deleteRow()	supprime la ligne courante
rowDeleted()	indique si la ligne courante est supprimée
moveToInsertRow()	permet de créer une nouvelle ligne dans l'ensemble de résultat

inserRow()	permet de valider la création de la ligne

Pour réaliser une mise à jour dans la ligne courante désignée par le curseur, il faut utiliser une des méthodes updateXXX() sur chacun des champs à modifier. Une fois toutes les modifications faites dans une ligne, il faut appeler la méthode updateRow() pour reporter ces modifications dans la base de données car les méthodes updateXXX() ne font des mises à jour que dans le jeu de résultats. Les mises à jour sont perdues si un changement de ligne intervient avant l'appel à la méthode updateRow().

La méthode cancelRowUpdates() permet d'annuler toutes les modifications faites dans la ligne. L'appel à cette méthode doit être effectué avant l'appel à la méthode updateRow().

Pour insérer une nouvelle ligne dans le jeu de résultat, il faut tout d'abord appeler la méthode moveToInsertRow(). Cette méthode déplace le curseur vers un buffer dédié à la création d'une nouvelle ligne. Il faut alimenter chacun des champs nécessaires dans cette nouvelle ligne. Pour valider la création de cette nouvelle ligne, il faut appeler la méthode insertRow().

Pour supprimer la ligne courante, il faut appeler la méthode deleteRow(). Cette méthode agit sur le jeu de résultats et sur la base de données.

40.12.2. Les mises à jour de masse (Batch Updates)

JDBC 2.0 permet de réaliser des mises à jour de masse en regroupant plusieurs traitements pour les envoyer en une seule fois au SGBD. Ceci permet d'améliorer les performances surtout si le nombre de traitements est important.

Cette fonctionnalité n'est pas obligatoirement supportée par le pilote. La méthode supportsBatchUpdate() de la classe DatabaseMetaData permet de savoir si elle est utilisable avec le pilote.

Plusieurs méthodes ont été ajoutées à l'interface Statement pour pouvoir utiliser les mises à jour de masse :

Méthode	Rôle
void addBatch(String)	permet d'ajouter une chaîne contenant une requête SQL
int[] executeBatch()	permet d'exécuter toutes les requêtes. Elle renvoie un tableau d'entier qui contient pour chaque requête, le nombre de mises à jour effectuées.
void clearBatch()	supprime toutes les requêtes stockées

Lors de l'utilisation de batchupdate, il est préférable de positionner l'attribut autocommit à false afin de faciliter la gestion des transactions et le traitement d'une erreur dans l'exécution d'un ou plusieurs traitements.

Exemple (code jdbc 2.0) :

```
connection.setAutoCommit(false);
Statement statement = connection.createStatement();

for(int i=0; i<10 ; i++) {
   statement.addBatch("INSERT INTO personne VALUES('nom"+i+"','prenom"+i+"')");
}
statement.executeBatch();
```

Une exception particulière peut être levée en plus de l'exception SQLException lors de l'exécution d'une mise à jour de masse. L'exception SQLException est levée si une requête SQL d'interrogation doit être exécutée (requête de type SELECT). L'exception BatchUpdateException est levée si une des requêtes de mise à jour échoue.

L'exception BatchUpdateException possède une méthode getUpdateCounts() qui renvoie un tableau d'entier qui contient le nombre d'occurrences impactées par chaque requête réussie.

40.12.3. Le package javax.sql

Ce package est une extension à l'API JDBC qui propose des fonctionnalités pour les développements coté serveur. C'est pour cette raison, que cette extension est uniquement intégrée à J2EE.

Les principales fonctionnalités proposées sont :

- une nouvelle interface pour assurer la connexion : l'interface DataSource
- les pools de connexions
- les transactions distribuées
- l'API Rowset

DataSource et Rowset peuvent être utilisées directement. Les pools de connexions et les transactions distribuées sont utilisés par une implémentation dans les serveurs d'applications pour fournir ces services.

40.12.4. La classe DataSource

La classe DataSource propose de fournir une meilleure alternative à la classe DriverManager pour assurer la connexion à une base de données.

Elle représente une connexion physique à une base de données. Les fournisseurs de pilotes proposent une implémentation de l'interface DataSource.

L'utilisation d'un objet DataSource est obligatoire pour pouvoir utiliser un pool de connexion et les transactions distribuées. Une fois créé un objet de type DataSource doit être enregistré dans un service de nommage. Il suffit alors d'utiliser JNDI pour obtenir une instance de classe DataSource.

Exemple :

```
...
Context ctx = new InitialContext();
DataSource ds = (DataSource) ctx.lookup("jdbc/applicationDB");
Connection con = ds.getConnection("admin", "mpadmin");
...
```

La suite de cette section sera développée dans une version future de ce document

40.12.5. Les pools de connexion

Un pool de connexions permet de maintenir un ensemble de connexions établies vers une base de données qui sont réutilisables. L'établissement d'une connexion est très couteux en ressources. L'intérêt du pool de connexions est de limiter le nombre de ces créations et ainsi d'améliorer les performances surtout si le nombre de connexions est important.

La suite de cette section sera développée dans une version future de ce document

40.12.6. Les transactions distribuées

Les connexions obtenues à partir d'un objet DataSource peuvent participer à une transaction distribuée.

La suite de cette section sera développée dans une version future de ce document

40.12.7. L'API RowSet

L'interface javax.sql.Rowset définie des objets qui permettent de manipuler les données d'une base de données.

Pour utiliser l'interface RowSet, il est nécessaire d'avoir une implémentation : l'implémentation de référence de Sun, une implémentation d'un tiers (par exemple le fournisseur du pilote JDBC) ou développer sa propre implémentation.

L'implémentation d'un RowSet peut être de deux types :

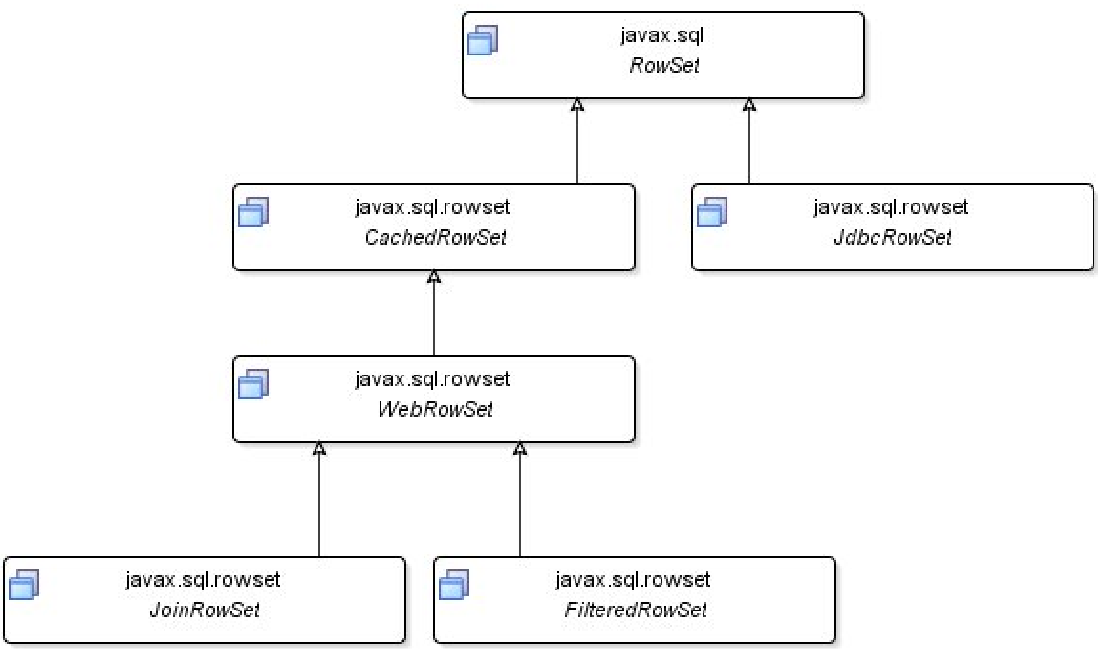
- connecté à la base de données durant toute sa durée de vie
- déconnecté de la base après avoir récupéré des données dans la base pour permettre une manipulation des données en mode déconnecté. Les modifications peuvent alors être reportées dans la base lors d'une reconnexion ultérieure.

Un RowSet de type déconnecté doit posséder un objet de type RowSetReader pour permettre la lecture des données et un objet de type RowSetWriter pour permettre l'enregistrement des données.

Avant Java 5, l'implémentation de référence de Rowset proposée par Sun est téléchargeable séparément à l'url
http://java.sun.com/products/jdbc/download.html#rowset1_0

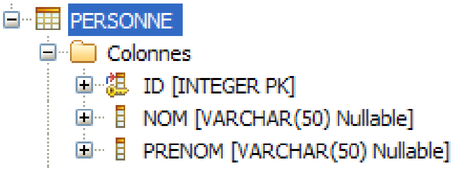
Java 5 fourni en standard une implémentation de référence des interfaces filles de l'interface RowSet définies dans la JSR 114 :

- JDBCRowSet : manipuler les données en mode connecté
- CachedRowSet : manipuler les données d'une source de données en mode déconnecté (les données sont stockées dans l'objet)
- WebRowSet : permet la lecture et l'écriture des données au format XML (hérite de CachedRowSet)
- FilteredRowSet : permet de faire des filtres (hérite de WebRowSet)
- JoinRowSet : permet de faire des jointures (hérite de WebRowSet) avec des objets implémentant l'interface Joinable



Ces interfaces filles sont définies dans le package javax.sql. Les implémentations sont nommées du nom de l'interface suivi de impl : elles sont regroupées dans le package javax.sql.rowset.

Les exemples de cette section utilisent une base de données JavaDB en mode embeded ou client/server selon les besoins. La table utilisée est composée de 3 champs :



La table personne contient 3 occurrences

ID [INTEGER]	NOM [VARCHAR(50)]	PRENOM [VARCHAR(50)]
1	nom1	prenom1
2	nom2	prenom2
3	nom3	prenom3

40.12.7.1. L'interface RowSet

Un RowSet est un objet qui encapsule les données d'une source de données. L'implémentation d'un RowSet est un Javabean. Un RowSet peut obtenir lui-même ces données en se connectant à la base de données.

L'interface RowSet est définie depuis la version 2.0 de l'API JDBC. Elle hérite de l'interface ResultSet : elle encapsule donc des données tabulaires et leur utilisation générale est similaire.

L'intérêt des objets de type RowSet est que ce sont des javabeans : ils gèrent donc des propriétés, sont sérialisables et peuvent mettre en oeuvre un mécanisme d'événements. Elle permet la mise en oeuvre de JDBC au travers d'un javabean.

Le fait que les RowSet soit des JavaBeans permet de les serialiser (pour des échanges à travers le réseau par exemple) ou de les utiliser directement avec d'autres Java Beans (avec les composants Swing dans une interface graphique par exemple).

Les implémentations de l'interface RowSet sont sérialisables ce qui facilite leur utilisation par rapport aux objets de type ResultSet qui ne le sont pas. Ils peuvent par exemple être utilisés par des EJB.

Cette interface propose un ensemble de propriétés pour permettre la connexion à une source de données. La propriété command contient la requête SQL qui permet d'obtenir les données. Ceci permet d'éviter la mise en oeuvre des différents objets de l'API JDBC (Connection et Statment notamment).

La méthode setURL() permet de préciser l'url JDBC utilisée lors de la connexion. Les méthodes setUsername() et setPassword() permettent fournir le nom du user et son mot de passe pour la connexion.

La méthode setCommand() permet de préciser la requête qui sera exécutée pour obtenir les données.

La méthode execute() permet des réaliser les traitements pour charger les données (connexion à la base de données, exécution de la requête, parcours des données et éventuellement fermeture de la connexion selon l'implémentation du RowSet).

Le parcours des données se fait de la même façon que pour un ResultSet sachant qu'il peut toujours se faire dans les deux sens selon le paramétrage du RowSet (utilisation des méthodes first(), last(), next() et previous()).

Un RowSet peut être rempli de deux façons :

- En lui fournissant les informations de connexion et la requête à exécuter
- En lui fournissant un objet de type ResultSet qui contient déjà les données issues de l'exécution d'une requête à la méthode populate()

Une fois rempli le RowSet peut toujours être parcouru dans les deux sens même si le pilote JDBC utilisé pour remplir les données ne permet pas cette fonctionnalité. La méthode size() permet de connaître le nombre d'occurrences contenues dans le RowSet.

Attention : lorsque le RowSet est rempli grâce à un ResultSet, il est nécessaire pour faire des modifications dans la table de la base de données de fournir au Rowset les informations de connexion et même la table concernée en utilisant la méthode setTableName().

Il est possible de préciser le niveau d'isolation de la transaction utilisée avec la connexion.

Exemple :

```
rs.setTransactionIsolation(
        Connection.TRANSACTION_READ_COMMITTED);
```

Les interfaces des spécifications de RowSet sont contenues dans le package javax.sql.rowset.
L'implémentation fournie avec le JDK est contenue dans le package com.sun.rowset : elle a été spécifiée par la JSR 114. Elle propose 5 RowSets standards : JdbcRowSet, CachedRowSet, WebRowSet, FilteredRowSet et JoinRowSet

Le JdbcRowSet fonctionne en mode connecté alors que CachedRowSet, WebRowSet, FilteredRowSet et JoinRowSet fonctionne en mode déconnecté.

40.12.7.2. L'interface JdbcRowSet

JdbcRowSet est un Rowset connecté qui encapsule un ResultSet.

Contrairement au ResultSet, JdbcRowSet permet d'encapsuler un ensemble de données et de proposer un parcours des données dans les deux sens même si l'implémentation du ResultSet utilisé pour le remplir ne le permet pas.

JdbcRowSet peut donc être parcourable dans les deux sens et peut être mis à jour.

Java 5 fournie une implémentation de cette interface avec la classe com.sun.rowset.JdbcRowSetImpl

La classe JdbcRowSetImpl possède deux constructeurs :

- sans paramètre
- avec un objet de type ResultSet en paramètre

En utilisant le constructeur sans paramètre, il est nécessaire d'utiliser les méthodes utiles à la configuration de la connexion et de la requête à exécuter.

Exemple (Java 5) :

```
package com.jmdoudoux.test.rowset;

import java.sql.ResultSet;

import javax.sql.rowset.JdbcRowSet;

import com.sun.rowset.JdbcRowSetImpl;

public class TestJdbcRowSet {

  public static void main(String[] args) {
    JdbcRowSet rs;

    try {
      Class.forName("org.apache.derby.jdbc.EmbeddedDriver");

      rs = new JdbcRowSetImpl();
      rs.setUrl("jdbc:derby:C:/Program Files/Java/jdk1.6.0/db/MaBaseDeTest");
      rs.setUsername("APP");
      rs.setPassword("");
      rs.setCommand("SELECT * FROM PERSONNE");
      rs.setConcurrency(ResultSet.CONCUR_READ_ONLY);
      rs.execute();

      while (rs.next()) {
        System.out.println("nom : "
            + rs.getString("nom")
            + ", prenom : "
            + rs.getString("prenom"));
      }

      rs.close();

    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Il est possible d'utiliser des paramètres dans la requête passée en paramètre de la méthode setCommand(). Chacun des paramètres est défini avec le caractère « ? ». La valeur de chaque paramètre est fournie en utilisant une des méthodes setXXX() qui attend en paramètre l'index du paramètre et sa valeur.

Exemple (Java 5) :

```
package com.jmdoudoux.test.rowset;

import java.sql.ResultSet;

import javax.sql.rowset.JdbcRowSet;

import com.sun.rowset.JdbcRowSetImpl;

public class TestJdbcRowSet2 {

  public static void main(String[] args) {
    JdbcRowSet rs;

    try {
      Class.forName("org.apache.derby.jdbc.EmbeddedDriver");

      rs = new JdbcRowSetImpl();
      rs.setUrl("jdbc:derby:C:/Program Files/Java/jdk1.6.0/db/MaBaseDeTest");
      rs.setUsername("APP");
      rs.setPassword("");
```

```
      rs.setCommand("SELECT * FROM PERSONNE where id > ?");
      rs.setInt(1, 2);
      rs.setConcurrency(ResultSet.CONCUR_READ_ONLY);
      rs.execute();

      while (rs.next()) {
        System.out.println("nom : "
            + rs.getString("nom")
            + ", prenom : "
            + rs.getString("prenom"));
      }

      rs.close();

    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

En utilisant le constructeur attendant en paramètre un objet de type ResultSet, l'instance obtenue encapsule les données du ResultSet. Ces données peuvent être parcourues dans les deux sens et sont modifiables.

Exemple (Java 5) :

```
package com.jmdoudoux.test.rowset;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;

import javax.sql.rowset.JdbcRowSet;

import com.sun.rowset.JdbcRowSetImpl;

public class TestJdbcRowSet3 {

  public static void main(String[] args) {
    JdbcRowSet rs;

    try {
      Connection conn = null;
      Statement stmt = null;

      Class.forName("org.apache.derby.jdbc.EmbeddedDriver");

      conn = DriverManager.getConnection(
            "jdbc:derby:C:/Program Files/Java/jdk1.6.0/db/MaBaseDeTest;user=APP");
      stmt = conn.createStatement();
      ResultSet resultSet = stmt.executeQuery("select * from personne");

      rs = new JdbcRowSetImpl(resultSet);

      while (rs.next()) {
        System.out.println("nom : "
            + rs.getString("nom")
            + ", prenom : "
            + rs.getString("prenom"));
      }

      rs.close();

    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Les données encapsulées dans le RowSet peuvent être mises à jour en fournissant la valeur ResultSet.CONCUR_UPDATABLE à la méthode setConcurrency(). Des méthodes updateXXX() héritées de la classe ResultSet permettent de mettre à jour une donnée en fonction de son type.

La méthode updateRow() permet de demander la mise à jour des données dans le RowSet.

La méthode commit() permet de demander la répercussion des modifications dans la base de données.

Exemple (Java 5) :

```
package com.jmdoudoux.test.rowset;

import java.sql.ResultSet;

import javax.sql.rowset.JdbcRowSet;

import com.sun.rowset.JdbcRowSetImpl;

public class TestJdbcRowSet4 {

  public static void main(String[] args) {
    JdbcRowSet rs;

    try {
      Class.forName("org.apache.derby.jdbc.EmbeddedDriver");

      rs = new JdbcRowSetImpl();
      rs.setUrl("jdbc:derby:C:/Program Files/Java/jdk1.6.0/db/MaBaseDeTest");
      rs.setUsername("APP");
      rs.setPassword("");
      rs.setCommand("SELECT * FROM PERSONNE");
      rs.setConcurrency(ResultSet.CONCUR_UPDATABLE);
      rs.execute();

      rs.absolute(2);
      rs.updateString("nom", "nom2 modifie");
      rs.updateRow();
      rs.commit();

      rs.close();

    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

40.12.7.3. L'interface CachedRowSet

L'interface CachedRowSet définie un RowSet déconnecté : la connexion à la base de données n'est maintenue que pour récupérer toutes les données. Toutes ces données sont stockées dans l'objet et la connexion est fermée. Il est alors possible de manipuler ces données (consultation et mise à jour). Les modifications peuvent alors être rendues persistantes en utilisant une nouvelle connexion dédiée à cette tâche.

Ceci peut permet de réduire les ressources réseaux et serveurs mais introduit généralement des problématiques de synchronisation des mises à jour.

L'implémentation standard de l'interface CachedRowSet est proposée par la classe com.sun.rowset.CachedRowSetImpl. Cet objet maintient l'état des données qu'il encapsule en mémoire. Il a simplement besoin de la connexion pour remplir ces données et plus tard au moment de rendre les modifications sur ces données persistantes.

Exemple (Java 5) :

```
package com.jmdoudoux.test.rowset;
```

```
import java.sql.ResultSet;

import javax.sql.rowset.CachedRowSet;

import com.sun.rowset.CachedRowSetImpl;

public class TestCachedRowSet {

  public static void main(String[] args) {
    CachedRowSet rs;

    try {
      Class.forName("org.apache.derby.jdbc.EmbeddedDriver");

      rs = new CachedRowSetImpl();
      rs.setUrl("jdbc:derby:C:/Program Files/Java/jdk1.6.0/db/MaBaseDeTest");
      rs.setCommand("SELECT * FROM PERSONNE");
      rs.setUsername("APP");
      rs.setPassword("");
      rs.setConcurrency(ResultSet.CONCUR_READ_ONLY);
      rs.execute();

      while (rs.next()) {
        System.out.println("nom : " + rs.getString("nom"));
      }

      rs.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

La méthode populate() permet de remplir le rowSet avec les données d'un ResultSet.

Ce premier exemple n'est pas pertinent car il aurait été plus efficace d'utiliser directement le ResultSet. Par contre, le CachedRowSet devient intéressant dès qu'il faut faire des mises à jour sans être connecté à la base de données

Les mises à jour sont faites uniquement dans l'objet CachedRowSet. Pour reporter ces modifications dans la base de données, il faut utiliser la méthode acceptChanges(). Lors de l'appel à cette méthode, l'objet CachedRowSet se reconnecte à la base de données et effectue les mises à jour.

Exemple (Java 5) :

```
package com.jmdoudoux.test.rowset;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;

import javax.sql.rowset.CachedRowSet;

import com.sun.rowset.CachedRowSetImpl;

public class TestCachedRowSet3 {

  public static void main(String[] args) {
    CachedRowSet rs;

    try {

      Connection conn = null;
      Statement stmt = null;

      Class.forName("org.apache.derby.jdbc.EmbeddedDriver");

      conn = DriverManager.getConnection(
            "jdbc:derby:C:/Program Files/Java/jdk1.6.0/db/MaBaseDeTest;user=APP");
      stmt = conn.createStatement();
```

```
      ResultSet resultSet = stmt.executeQuery"select * from personne");

      rs = new CachedRowSetImpl();
      rs.populate(resultSet);

      rs.absolute(2);
      rs.updateString("nom", "nom2");
      rs.updateRow();

      rs.acceptChanges(conn);

      rs.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

La propriété COMMIT_ON_ACCEPT_CHANGES est un booléen qui permet de préciser si un commit est réalisé automatiquement à la fin de la méthode acceptChanges(). La valeur par défaut est true. Si sa valeur est false, il faut explicitement faire appel à la méthode commit() pour valider la transaction.

Il est tout à fait possible que les données dans la base soient modifiées entre la récupération des données et leur mise à jour dans la base de données. Avant chaque mise à jour, CachedRowSet vérifie les données courantes dans la base avec leur valeur initiale lors du remplissage des données. Si une différence est détectée alors une exception de type SyncProviderException est levée.

Exemple (Java 5) :

```
package com.jmdoudoux.test.rowset;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;

import javax.sql.rowset.CachedRowSet;

import com.sun.rowset.CachedRowSetImpl;

public class TestCachedRowSet3 {

  public static void main(String[] args) {
    CachedRowSet rs;

    try {

      Connection conn = null;
      Statement stmt = null;

      Class.forName("org.apache.derby.jdbc.ClientDriver");

      java.util.Properties props = new java.util.Properties();
      props.put("user","APP");
      props.put("password","APP");
      conn = DriverManager.getConnection("jdbc:derby://localhost:1527/MaBaseDeTest", props);

      stmt = conn.createStatement();
      ResultSet resultSet = stmt.executeQuery("select * from personne");

      rs = new CachedRowSetImpl();
      rs.populate(resultSet);

      System.out.println("debut attente");
      Thread.sleep(60000);
      // mise à jour de l'occurence dans la base de données par un outil externe
      System.out.println("fin attente");

      rs.absolute(2);
      rs.updateString("nom", "nom2");
```

```
      rs.updateRow();

      rs.acceptChanges(conn);

      rs.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
debut attente
fin attente
javax.sql.rowset.spi.SyncProviderException: 3 conflicts while synchronizing
        at com.sun.rowset.internal.CachedRowSetWriter.writeData(CachedRowSetWriter.java:373)
        at com.sun.rowset.CachedRowSetImpl.acceptChanges(CachedRowSetImpl.java:862)
        at com.sun.rowset.CachedRowSetImpl.acceptChanges(CachedRowSetImpl.java:921)
        at com.jmdoudoux.test.rowset.TestCachedRowSet3.main(TestCachedRowSet3.java:43)
```

Le CachedRowSet propose un mécanisme pour gérer ce cas de figure. Ce mécanisme impose de préciser au CachedRowSet la ou les colonnes qui représentent la clé ceci afin de lui permettre de faire correspondre ces occurrences avec celles de la base de données : c'est la méthode setKeyColumns() qui attend en paramètre un tableau entier contenant les index des colonnes.

Remarque : l'index de colonnes utilisées dans un CachedRowSet commence à 1 à non à 0.

Le traitement des conflits est à faire dans le traitement de l'exception de type SyncProviderException. Cette exception propose la méthode getSyncResolver() qui renvoie un objet de type SyncResolver.

L'objet de type SyncResolver va permet d'obtenir les conflits détectés et les résoudre en fonction des besoins. L'interface SyncResolver définit plusieurs méthodes :

Méthode	Rôle
Object getConflictValue()	Retourne la valeur dans la base de données de l'occurrence courante du SyncResolver pour la colonne fournie en paramètre (index ou nom selon la surcharge utilisée). La valeur retournée est null pour une colonne qui n'est pas en conflit.
int getStatus()	Renvoie un entier qui précise le type d'opération tentée sur la base de données : DELETE_ROW_CONFLICT, INSERT_ROW_CONFLICT, UPDATE_ROW_CONFLICT ou NO_ROW_CONFLICT
boolean nextConflict()	Se déplace sur le prochain conflit s'il existe et renvoie un booléen si le déplacement a eu lieu
boolean previousConflict()	Se déplace sur le conflit précédent s'il existe et renvoie un booléen si le déplacement a eu lieu
void setResolvedValue()	Permet de définir la valeur dans la base de données de l'occurrence courante du SyncResolver pour la colonne fournie en paramètre (index ou nom selon la surcharge utilisée)

Chaque fournisseur propose sa propre implémentation de SyncProvider. Les exemples de cette section utilisent l'implémentation de référence fournie avec le JDK à partir de la version 5.0. Cette implémentation propose un mode de gestion optimiste des accès concurrents (aucun verrou n'est posé sur les occurrences dans la base de données).

Il faut réaliser une itération sur les conflits en utilisant la méthode nextConflict().

La méthode getStatus() permet de connaître le type de mise jour tentée sur la base de données

La méthode getRow() héritée de l'interface ResultSet permet de connaître l'index de l'occurrence concernée par le conflit. Ceci permet de se déplacer dans le RowSet pour obtenir les nouvelles valeurs.

La méthode getConflictValue() est utilisée dans une itération sur les colonnes pour déterminer celles qui sont en conflit : dans ce cas la valeur retournée est différente de null.

A partir de la nouvelle valeur, de la valeur courante dans la base de données et du type de mise à jour, les traitements doivent déterminer la valeur à mettre dans la base de données.

Cette valeur est fournie en utilisant la méthode setResolvedValue().

Exemple (Java 5) :

```
package com.jmdoudoux.test.rowset;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

import javax.sql.rowset.CachedRowSet;

import com.sun.rowset.CachedRowSetImpl;
import javax.sql.rowset.spi.SyncProviderException;
import javax.sql.rowset.spi.SyncResolver;

public class TestCachedRowSet4 {

  public static void main(String[] args) {
    CachedRowSet rs=null;

    try {

      Connection conn = null;
      Statement stmt = null;

      Class.forName("org.apache.derby.jdbc.ClientDriver");

      java.util.Properties props = new java.util.Properties();
      props.put("user", "APP");
      props.put("password", "APP");
      conn = DriverManager.getConnection(
          "jdbc:derby://localhost:1527/MaBaseDeTest", props);

      stmt = conn.createStatement();
      ResultSet resultSet = stmt.executeQuery("select * from personne");

      rs = new CachedRowSetImpl();
      rs.populate(resultSet);
      rs.setTableName("PERSONNE");
      // la première colonne compose la clé
      rs.setKeyColumns(new int[] { 1 });

      System.out.println("debut attente");
      Thread.sleep(60000);
      // mise à jour de l'occurrence dans la
          // base de données par un outil externe
      System.out.println("fin attente");

      rs.absolute(2);
      rs.updateString("nom", "nom2");
      rs.updateRow();

      rs.acceptChanges(conn);

      rs.close();
    } catch (SyncProviderException spe) {
      SyncResolver resolver = spe.getSyncResolver();

      try {
        while (resolver.nextConflict()) {
          if (resolver.getStatus() == SyncResolver.UPDATE_ROW_CONFLICT) {
            int row = resolver.getRow();
            rs.absolute(row);
```

```
            int nbColonne = rs.getMetaData().getColumnCount();
            for (int i = 1; i <= nbColonne; i++) {
              if (resolver.getConflictValue(i) != null) {
                Object valeur = rs.getObject(i);
                Object valeurResolver = resolver.getConflictValue(i);
                System.out.println("champ = "
                                  + rs.getMetaData().getColumnName(i)
                                  +" , Valeur = "+valeur+"
                                  , valeur dans la base="+valeurResolver);
                // Determiner la valeur à mettre dans la base
                // dans ce cas simplement la nouvelle valeur
                resolver.setResolvedValue(i,  valeur);
              }
            }
          }
        }
      } catch (SQLException e) {
        e.printStackTrace();
      }
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
debut attente
fin attente
champ = NOM , Valeur = nom2 , valeur dans la base=nom2 mod
```

L'interface CachedRowSet propose plusieurs méthodes pour annuler des mises à jour faites dans les données encapsulées (avant l'appel à la méthode acceptChanges()) :

Méthode	Rôle
void undoDelete()	Annule l'opération de suppression de l'occurrence courante
void undoInsert()	Annule l'opération d'insertion de l'occurrence courante
void undoUpdate()	Annule l'opération de modification de l'occurence
void restoreOriginal()	Remettre l'ensemble des données à leur valeur originale (toutes les modifications sont perdues) et remet le curseur avant la première occurrence

La méthode getOriginal() renvoie un ResultSet qui contient toutes les valeurs originales des données du RowSet.

Le stockage des données en mémoire rend le CachedRowSet inapproprié à une utilisation avec de gros volume de données. Dans ce cas, le CachedRowSet peut travailler en paginant sur des portions de données : l'ensemble des données est traité par page (une page contenant un certain nombre d'occurrences). La méthode setPageSize() permet de préciser le nombre maximum d'occurrences dans une page. La méthode nextPage() permet d'obtenir la page suivante. Ce mécanisme est particulièrement utile pour traiter de grosses quantités de données.

La méthode release() permet de supprimer toutes les données contenues dans le RowSet : attention son appel fait perdre toutes les modifications dans les données qui n'ont pas été reportées dans la base de données .

40.12.7.4. L'interface WebRowSet

WebRowSet possède la capacité de lire ou d'écrire le contenu du RowSet au format XML. Cette faculté lui permet d'être utilisé pour échanger des données non pas sous une forme sérialisée mais sous la forme d'un document XML (par exemple dans une requête HTTP ou SOAP).

Dans l'implémentation standard, le document XML respecte le schéma :
http://java.sun.com/xml/ns/jdbc/webrowset.xsd

Le contenu au format XML d'un WebRowSet peut être exporté dans un flux quelconque : par exemple, l'envoie du contenu XML d'un WebRowSet dans une réponse d'une servlet.

Le document XML issu d'un WebRowSet possède un noeud racine <webRowSet> qui possède trois noeuds fils :

- <properties> : contient les propriétés du WebRowSet notamment les paramètres de connexion sauf le user et mot de passe
- <metadata> : contient les méta-données du WebRowSet (configuration de chaque colonne)
- <data> : contient les données du WebRowSet

Chaque occurrence de données est stockée dans un tag <currentRow>. La valeur de chaque colonne est stockée dans un tag <columnValue>.

Les occurrences ajoutées sont stockées dans un tag <insertRow>.
Les occurrences modifiées sont stockées dans un tag <updateRow>. La valeur de chaque colonne modifiée est stockée dans un tag fils <updateValue>
Les occurrences supprimées sont stockées dans un tag <deleteRow>.

Exemple (Java 5) :

```
package com.jmdoudoux.test.rowset;

import java.sql.ResultSet;

import javax.sql.rowset.WebRowSet;
import com.sun.rowset.WebRowSetImpl;


public class TestWebRowSet {

  public static void main(String[] args) {
    WebRowSet rs;

    try {
      Class.forName("org.apache.derby.jdbc.ClientDriver");

      rs = new WebRowSetImpl();
      rs.setUrl("jdbc:derby://localhost:1527/MaBaseDeTest");
      rs.setCommand("SELECT * FROM PERSONNE");
      rs.setUsername("APP");
      rs.setPassword("APP");
      rs.setConcurrency(ResultSet.CONCUR_READ_ONLY);
      rs.execute();
      rs.writeXml(System.out);

      rs.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
<?xml  version="1.0"?>
<webRowSet
      xmlns="http://java.sun.com/xml/ns/jdbc"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://java.sun.com/xml/ns/jdbc
          http://java.sun.com/xml/ns/jdbc/webrowset.xsd">
  <properties>
    <command>SELECT * FROM  PERSONNE</command>
    <concurrency>1007</concurrency>
     <datasource><null/></datasource>
     <escape-processing>true</escape-processing>
     <fetch-direction>1000</fetch-direction>
```

```
    <fetch-size>0</fetch-size>
     <isolation-level>2</isolation-level>
    <key-columns>
    </key-columns>
    <map>
    </map>
     <max-field-size>0</max-field-size>
    <max-rows>0</max-rows>
     <query-timeout>0</query-timeout>
    <read-only>true</read-only>
     <rowset-type>ResultSet.TYPE_SCROLL_INSENSITIVE</rowset-type>
    <show-deleted>false</show-deleted>
     <table-name>PERSONNE</table-name>
     <url>jdbc:derby://localhost:1527/MaBaseDeTest</url>
    <sync-provider>
       <sync-provider-name>com.sun.rowset.providers.RIOptimisticProvider</sync-provider-name>
      <sync-provider-vendor>Sun  Microsystems Inc.</sync-provider-vendor>
       <sync-provider-version>1.0</sync-provider-version>
       <sync-provider-grade>2</sync-provider-grade>
       <data-source-lock>1</data-source-lock>
    </sync-provider>
  </properties>
  <metadata>
    <column-count>3</column-count>
    <column-definition>
       <column-index>1</column-index>
       <auto-increment>false</auto-increment>
       <case-sensitive>false</case-sensitive>
      <currency>false</currency>
      <nullable>0</nullable>
      <signed>true</signed>
      <searchable>true</searchable>
       <column-display-size>11</column-display-size>
       <column-label>ID</column-label>
      <column-name>ID</column-name>
       <schema-name>APP</schema-name>
       <column-precision>10</column-precision>
       <column-scale>0</column-scale>
       <table-name>PERSONNE</table-name>
      <catalog-name></catalog-name>
      <column-type>4</column-type>
       <column-type-name>INTEGER</column-type-name>
    </column-definition>
    <column-definition>
      <column-index>2</column-index>
       <auto-increment>false</auto-increment>
       <case-sensitive>true</case-sensitive>
      <currency>false</currency>
      <nullable>1</nullable>
      <signed>false</signed>
      <searchable>true</searchable>
      <column-display-size>50</column-display-size>
       <column-label>NOM</column-label>
       <column-name>NOM</column-name>
       <schema-name>APP</schema-name>
       <column-precision>50</column-precision>
       <column-scale>0</column-scale>
      <table-name>PERSONNE</table-name>
      <catalog-name></catalog-name>
      <column-type>12</column-type>
       <column-type-name>VARCHAR</column-type-name>
    </column-definition>
    <column-definition>
       <column-index>3</column-index>
      <auto-increment>false</auto-increment>
       <case-sensitive>true</case-sensitive>
      <currency>false</currency>
      <nullable>1</nullable>
      <signed>false</signed>
      <searchable>true</searchable>
       <column-display-size>50</column-display-size>
      <column-label>PRENOM</column-label>
       <column-name>PRENOM</column-name>
       <schema-name>APP</schema-name>
```

```
        <column-precision>50</column-precision>
        <column-scale>0</column-scale>
        <table-name>PERSONNE</table-name>
      <catalog-name></catalog-name>
      <column-type>12</column-type>
        <column-type-name>VARCHAR</column-type-name>
    </column-definition>
  </metadata>
  <data>
    <currentRow>
      <columnValue>1</columnValue>
        <columnValue>nom1</columnValue>
      <columnValue>prenom1</columnValue>
    </currentRow>
    <currentRow>
      <columnValue>2</columnValue>
        <columnValue>nom2</columnValue>
        <columnValue>prenom2</columnValue>
    </currentRow>
    <currentRow>
      <columnValue>3</columnValue>
      <columnValue>nom3</columnValue>
        <columnValue>prenom3</columnValue>
    </currentRow>
  </data>
</webRowSet>
```

La méthode readXml() permet de remplir l'objet WebRowSet avec un fichier XML par exemple précédemment créé grâce à la méthode writeXml().

40.12.7.5. L'interface FilteredRowSet

L'interface FilteredRowSet qui hérite de l'interface WebRowSet permet de mettre en oeuvre un filtre par programmation sans utiliser SQL.

FilteredRowSet est particulièrement utile car il permet de filtrer un ensemble de données sans avoir à effectuer une requête sur la base de données avec le filtre.

Le filtre est encapsulé dans une classe qui implémente l'interface Predicate. Dans cette classe, il faut redéfinir les méthodes evaluate() qui renvoie un booléen précisant si l'occurrence est conservée ou non par le filtre.

La méthode evaluate() acceptant en paramètre un objet de type RowSet est utilisée par l'objet FilteredRowSet lors du parcours de ces occurrences.

Les surcharges de la méthode evaluate() acceptant un objet et une colonne (par index ou par nom) sont utilisées par l'objet FilteredRowSet pour déterminer si une valeur d'une colonne correspond au filtre.

Exemple (Java 5) : ne conserver que les personnes dont le nom se termine par 2

```
package com.jmdoudoux.test.rowset;

import java.sql.SQLException;

import javax.sql.RowSet;
import javax.sql.rowset.Predicate;

public class PersonnePredicate implements Predicate {

  public boolean evaluate(Object value, int column) throws SQLException {
    // inutilisé dans cet exemple
    return false;
  }

  public boolean evaluate(Object value, String columnName) throws SQLException {
    // inutilisé dans cet exemple
```

```
    return false;
  }

  public boolean evaluate(RowSet rowset) {
    try {
      String nom = rowset.getString("nom");
      if (nom.endsWith("2")) {
        return true;
      } else {
        return false;
      }
    } catch (SQLException sqle) {
      return false;
    }
  }

}
```

Le filtre est précisé au FilteredRowSet en utilisant la méthode setFilter() qui attend en paramètre une instance de la classe Predicate.

Exemple (Java 5) :

```
package com.jmdoudoux.test.rowset;

import java.sql.ResultSet;

import javax.sql.rowset.FilteredRowSet;

import com.sun.rowset.FilteredRowSetImpl;

public class TestFilteredRowSet {

  public static void main(String[] args) {
    FilteredRowSet rs;

    try {
      Class.forName("org.apache.derby.jdbc.EmbeddedDriver");

      rs = new FilteredRowSetImpl();
      rs.setUrl("jdbc:derby:C:/Program Files/Java/jdk1.6.0/db/MaBaseDeTest");
      rs.setCommand("SELECT * FROM PERSONNE");
      rs.setUsername("APP");
      rs.setPassword("");
      rs.setConcurrency(ResultSet.CONCUR_READ_ONLY);
      rs.setFilter(new PersonnePredicate());
      rs.execute();

      while (rs.next()) {
        System.out.println("nom : " + rs.getString("nom"));
      }

      rs.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
nom : nom2
```

40.12.7.6. L'interface JoinRowSet

L'interface JoinRowSet qui hérite de l'interface WebRowSet permet de faire des jointures entre plusieurs instances de l'interface Joinable. Les interfaces qui héritent de Joinable sont : CachedRowSet, FilteredRowSet, JdbcRowSet, JoinRowSet, WebRowSet.

JoinRowSet peut être particulièrement utile si les données des RowSet qu'il encapsule appartiennent à des sources de données différentes

Pour utiliser un JoinRowSet, il faut en créer une instance et utiliser la méthode addRowSet() pour ajouter les instances de l'interface Joinable à utiliser dans la jointure. La méthode adddRowSet() possède plusieurs surcharges qui permettent de préciser l'instance de Joinable et la ou les clés utilisées lors de la jointure.

Exemple (Java 5) :

```
package com.jmdoudoux.test.rowset;

import java.sql.ResultSet;

import javax.sql.rowset.CachedRowSet;
import javax.sql.rowset.JoinRowSet;

import com.sun.rowset.CachedRowSetImpl;
import com.sun.rowset.JoinRowSetImpl;

public class TestJoinRowSetRowSet {

  public static void main(String[] args) {
    CachedRowSet rs1;
    CachedRowSet rs2;
    JoinRowSet rs;

    try {
      Class.forName("org.apache.derby.jdbc.EmbeddedDriver");

      rs1 = new CachedRowSetImpl();
      rs1.setUrl("jdbc:derby:C:/Program Files/Java/jdk1.6.0/db/MaBaseDeTest");
      rs1.setCommand("SELECT * FROM PERSONNE");
      rs1.setUsername("APP");
      rs1.setPassword("");
      rs1.setConcurrency(ResultSet.CONCUR_READ_ONLY);
      rs1.execute();

      rs2 = new CachedRowSetImpl();
      rs2.setUrl("jdbc:derby:C:/Program Files/Java/jdk1.6.0/db/MaBaseDeTest");
      rs2.setCommand("SELECT * FROM ADRESSE");
      rs2.setUsername("APP");
      rs2.setPassword("");
      rs2.setConcurrency(ResultSet.CONCUR_READ_ONLY);
      rs2.execute();

      rs = new JoinRowSetImpl();
      rs.addRowSet(rs1,1);
      rs.addRowSet(rs2,1);

      while (rs.next()) {
        System.out.println("nom : " + rs.getString("nom")+", rue : " + rs.getString("rue"));
      }

      rs.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
nom : nom3, rue : rue3
nom : nom2, rue : rue2
```

```
nom : nom1, rue : rue1
```

La méthode setJoinType() permet de préciser le type de jointure à effectuer en utilisant les constantes définies dans l'interface JoinRowSet : CROSS_JOIN, FULL_JOIN, INNER_JOIN (par défaut), LEFT_OUTER_JOIN et RIGHT_OUTER_JOIN. Les implémentations n'ont pas d'obligation à supporter tous les types de jointures : l'utilisation d'un type de jointure non supporté par l'implémentation lève une exception de type SQLException.

40.12.7.7. L'utilisation des événements

L'interface RowSetListener permet de gérer certains événements d'un RowSet. Le modèle d'événement des Javabeans est mis en oeuvre ou travers de ce listener de type RowSetListener et d'un événement de type RowSetEvent.

Les méthodes addRowSetListener() et removeRowSetListener() de l'interface RowSet permettent respectivement d'enregistrer et de supprimer un listener

L'interface RowSetListener définie trois méthodes :

- cursorMoved() : appelée lorsque le curseur de parcours des données change
- rowChanged() : appelée lorsqu'une donnée est modifiée
- rowSetChanged() : appelée lorsque l'ensemble des données est modifié

Exemple (Java 5) :

```
package com.jmdoudoux.test.rowset;

import java.sql.ResultSet;

import javax.sql.RowSetEvent;
import javax.sql.RowSetListener;
import javax.sql.rowset.JdbcRowSet;

import com.sun.rowset.JdbcRowSetImpl;

public class TestRowSetListener {

  public static void main(String[] args) {
    JdbcRowSet rs;

    try {
      Class.forName("org.apache.derby.jdbc.ClientDriver");

      rs = new JdbcRowSetImpl();
      rs.setUrl("jdbc:derby://localhost:1527/MaBaseDeTest");
      rs.setCommand("SELECT * FROM PERSONNE");
      rs.setUsername("APP");
      rs.setPassword("APP");
      rs.setConcurrency(ResultSet.CONCUR_READ_ONLY);
      rs.addRowSetListener(new RowSetListener() {
        public void cursorMoved(RowSetEvent event) {
          System.out.println("L'evenement cursorMoved est emis");
        }

        public void rowChanged(RowSetEvent event) {
          System.out.println("L'evenement rowChanged est emi"");
        }

        public void rowSetChanged(RowSetEvent event) {
          System.out.println("L'evenement rowSetChanged est emis");
        }

      });
      rs.execute();

      while (rs.next())
        System.out.println("nom : " + rs.getString("nom"));
```

```
      rs.close();

      rs.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  public TestRowSetListener() {
    JdbcRowSet rs;

    try {
      Class.forName("oracle.jdbc.driver.OracleDriver");

      rs = new JdbcRowSetImpl();
      rs.setUrl("jdbc:oracle:thin:@localhost:1521:test");
      rs.setCommand("SELECT * FROM article");
      rs.setUsername("test");
      rs.setPassword("test");
      rs.setConcurrency(ResultSet.CONCUR_READ_ONLY);
      rs.addRowSetListener(new RowSetListener() {
        public void cursorMoved(RowSetEvent event) {
          System.out.println("L'evenement cursorMoved est emis");
        }

        public void rowChanged(RowSetEvent event) {
          System.out.println("L'evenement rowChanged est emis");
        }

        public void rowSetChanged(RowSetEvent event) {
          System.out.println("L'evenement rowSetChanged est emis");
        }

      });
      rs.execute();

      while (rs.next())
        System.out.println("libelle : " + rs.getString("libelle"));

      rs.close();

    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
L'evenement rowSetChanged est emis
L'evenement cursorMoved est emis
nom : nom1
L'evenement cursorMoved est emis
nom : nom2
L'evenement cursorMoved est emis
nom : nom3
L'evenement cursorMoved est emis
```

40.13. JDBC 3.0

Les spécifications de l'API JDBC version 3.0, disponible à depuis mai 2002, sont issues des travaux de la JSR 0054 et sont directement intégrées dans la plate-forme J2SE 1.4.

Ces spécifications ont été développées en tenant compte de plusieurs points : conserver une compatibilité avec la version précédente de l'API, assurer une meilleure interaction avec la technologie JCA, le support de SQL 99, ...

Cette version propose plusieurs améliorations dont les savepoints, le support de SQL 99, la récupération des identifiants générés, ... détaillées dans les sections suivantes.

JDBC n'est qu'une spécification : l'implémentation réalisée au travers des pilotes peut proposer tout ou uniquement une partie de ces fonctionnalités.

40.13.1. Le nommage des paramètres d'un objet de type CallableStament

Avant la version 3.0, lors de l'utilisation d'une instance de l'interface CallableStatement, pour assigner une valeur à un paramètre, il fallait obligatoirement utiliser son index. Il est dorénavant possible d'utiliser un nom pour un paramètre et d'utiliser ce nom pour mettre à jour sa valeur.

L'interface CallableStatement c'est vu rajouter des surcharges des méthodes getXXX() et setXXX() attendant en premier paramètre une chaîne de caractères qui va contenir le nom du paramètre.

Cette fonctionnalité est intéressante notamment pour l'appel de procédure stockée qui possède des valeurs par défaut pour certains paramètres. Il est ainsi possible de ne fournir que les valeurs voulues lors de l'appel.

40.13.2. Les types java.sql.Types.DATALINK et java.sql.Types.BOOLEAN

Deux nouveaux types sont supportés : java.sql.Types.DATALINK pour des url vers des ressources externes et java.sql.Types.BOOLEAN pour les booléens. Les valeurs d'une donnée de ces types sont obtenues en utilisant respectivement les méthodes getURL() et getBoolean() de la classe ResultSet.

40.13.3. L'obtention des valeurs générées automatiquement lors d'une insertion

La plupart des bases de données relationnelles proposent des fonctionnalités pour permettre la génération d'une valeur, généralement auto incrémentée dans un champ d'une base de données, permettant la génération d'un identifiant unique. Ceci est très pratique pour définir un champ qui sera la clé primaire d'une table. Cependant avant la version 3.0 de JDBC, il était nécessaire d'effectuer une lecture après l'insertion des données.

Ceci pose souvent des problèmes notamment pour arriver à utiliser une clause where dans la requête d'interrogation qui soit sûre de renvoyer les données de la ligne insérée. De plus, cela impose de réaliser une opération supplémentaire sur la base de données.

Il est maintenant possible d'obtenir facilement la valeur d'un identifiant généré par la base de données lors de l'insertion d'une nouvelle occurrence dans une table. Attention, le support de cette fonctionnalité par le pilote est optionnel.

Il suffit de préciser lors de l'appel à la méthode executeUpdate() de l'interface Statement la valeur Statement.RETURN_GENERATED_KEYS au paramètre autoGeneratedKeys de type int.

Pour obtenir la valeur de la clé ou des clés générées, il suffit d'appeler la méthode getGeneratedKeys() de l'instance de l'interface Statement utilisée pour exécuter la mise à jour : le ResultSet retourné par cette méthode contient un champ pour chaque champ généré par la base de données.

Exemple :

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import java.sql.Statement;

public class TestJdbc101 {

  public static void main(java.lang.String[] args) {
```

```
    Connection con = null;
    Statement stmt = null;
    ResultSet resultats = null;
    String requete = "";

    // chargement du pilote
    try {
      Class.forName("com.mysql.jdbc.Driver").newInstance();
    } catch (Exception e) {
      e.printStackTrace();
      System.exit(99);
    }

    try {
      String DBurl = "jdbc:mysql://localhost:3306/testjava";
      con = DriverManager.getConnection(DBurl);
      stmt = con.createStatement();

      stmt.executeUpdate(
          "INSERT INTO personne (nom, prenom, taille) "
              + "values ('nom1', 'prenom1', 174)",
            Statement.RETURN_GENERATED_KEYS);
      int idGenere = -1;
      resultats = stmt.getGeneratedKeys();
      if (resultats.next()) {
        idGenere = resultats.getInt(1);
      } else {
        System.out.println("Impossible d'obtenir la valeur generee");
      }
      resultats.close();
      resultats = null;
      System.out.println("valeur id genere = " + idGenere);

    } catch (SQLException e) {
      e.printStackTrace();
    } finally {
      if (resultats != null) {
        try {
          resultats.close();
        } catch (SQLException ex) {
        }
      }
      if (stmt != null) {
        try {
          stmt.close();
        } catch (SQLException ex) {
        }
      }
      if (con != null) {
        try {
          con.close();
        } catch (SQLException ex) {
        }
      }
    }

    System.exit(0);
  }

}
```

Résultat :

```
valeur id genere = 1
```

40.13.4. Le support des points de sauvegarde (savepoint)

Pour utiliser les transactions, il est nécessaire de demander la désactivation du mode auto-commit de la connexion. Il faut appeler la méthode setAutoCommit() avec le paramètre false de l'instance de la classe Connection qui encapsule la connexion à la base de données.

La transaction peut alors être validée ou annulée en totalité avec respectivement les méthodes commit() et rollback().

Avant la version 3.0 de JDBC, il n'est possible que de valider toutes les opérations ou d'annuler toutes les opérations de la transaction. Il n'est pas possible de réaliser des validations ou des annulations d'un sous ensemble d'opérations de la transaction.

Avec la version 3.0 de JDBC, les savepoints permettent de définir des points nommés entre l'exécution de deux opérations de la transaction. Ce savepoint peut être considéré comme un marqueur. Toutes les opérations réalisées depuis la définition de ce marqueur peuvent être annulées sans que les opérations réalisées avant le marqueur ne soient annulées.

Pour définir un savepoint, il suffit d'appeler la méthode setSavePoint() de la classe Connection. Cette méthode renvoie un objet de type Savepoint qu'il faut passer en paramètre de la méthode rollback() pour annuler les opérations réalisées depuis la définition du savepoint.

Exemple :

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import java.sql.Savepoint;
import java.sql.Statement;

public class TestJdbc102 {

  public static void main(java.lang.String[] args) {
    Connection con = null;
    Statement stmt = null;
    ResultSet resultats = null;
    String requete = "";

    // chargement du pilote
    try {
      Class.forName("com.mysql.jdbc.Driver").newInstance();
    } catch (Exception e) {
      e.printStackTrace();
      System.exit(99);
    }

    try {
      String DBurl = "jdbc:mysql://localhost:3306/test";
      con = DriverManager.getConnection(DBurl);
      stmt = con.createStatement();

      con.setAutoCommit(false);
      con.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);

      stmt.executeUpdate("UPDATE personne set nom = 'nom1 modif1' where id=1");
      Savepoint svpt = con.setSavepoint("savepoint_1");
      stmt.executeUpdate("UPDATE personne set nom = 'nom1 modif2' where id=1");
      con.rollback(svpt);
      con.commit();

      // creation et execution de la requête
      requete = "SELECT * FROM personne";
      stmt = con.createStatement();
      resultats = stmt.executeQuery(requete);

      ResultSetMetaData rsmd = resultats.getMetaData();
      int nbCols = rsmd.getColumnCount();
      boolean encore = resultats.next();
      while (encore) {
        for (int i = 1; i <= nbCols; i++)
```

```
          System.out.print(resultats.getString(i) + " ");
        System.out.println();
        encore = resultats.next();
      }

      resultats.close();
      resultats = null;

    } catch (SQLException e) {
      e.printStackTrace();
      if (con != null) {
        try {
        con.rollback();
        } catch (SQLException ex) {
        }
      }
    } finally {
      if (resultats != null) {
        try {
          resultats.close();
        } catch (SQLException ex) {
        }
      }
      if (stmt != null) {
        try {
          stmt.close();
        } catch (SQLException ex) {
        }
      }
      if (con != null) {
        try {
          con.close();
        } catch (SQLException ex) {
        }
      }
    }

    System.exit(0);
  }
}
```

Dans l'exemple ci dessus, seule la première mise à jour est effective suite au commit de la transaction.

40.13.5. Le pool d'objets PreparedStatements

Il est maintenant possible d'utiliser un pool d'objets de type PreparedStatement. Cette mise en pool est transparente pour le développeur car elle est gérée par le pool de connexion.

Lors de la suppression d'un objet PreparedStatement, celui-ci est mis dans le pool pour permettre une réutilisation et ainsi évite une recompilation d'un nouvel objet de ce type.

Ceci permet d'éviter la répétition des traitements coûteux effectués lors de la création d'un objet PreparedStatement (vérification et optimisation de la requête par la base de données).

Un pilote compatible avec la version 3.0 de JDBC va ainsi mettre en place un pool pour ces objets : à leur fermeture, les objets sont remis dans le pool. Lorsque le PreparedStatement est réutilisé, l'objet est repris du pool plutôt que recréé.

40.13.6. La définition de propriétés pour les pools de connexions

La version 3.0 de JDBC propose un contrôle plus précis sur les paramètres du pool de connexion tel que la taille du pool, le nombre minimum et maximum de connexion qu'il contient, ...

L'utilisation de ces propriétés peut améliorer les performances sans modification dans le code qui met en oeuvre l'API JDBC puisqu'ils affectent des mécanismes transparents pour le développeur et qu'il n'est pas recommandé de modifier ces paramètres via l'API (il est préférable de les configurer au travers du serveur d'applications).

Ceci permet aussi de standardiser ces propriétés et de rendre la configuration moins dépendante des fournisseurs de pilote.

Propriété	Description
maxStatements	Précise le nombre maximum de statements gérés par le pool La valeur 0 indique une désactivation du mécanisme de mise en pool
initialPoolSize	Précise le nombre de connexions créées par le pool à sa création
minPoolSize	Précise le nombre de connexions minimum géré par le pool. La valeur 0 précise que les connexions seront créées en fonction des besoins.
maxPoolSize	Précise le nombre maximum de connexions gérées par le pool. La valeur 0 indique qu'il n'y a pas de maximum.
maxIdleTime	Précise la durée en secondes avant qu'une connexion du pool inutilisée ne soit fermée. La valeur 0 précise qu'il n'y pas de durée.

40.13.7. L'ajout de metadata pour obtenir la liste des types de données supportés

La méthode getType Info() permet d'obtenir un ResultSet qui contient la liste des types de données supportés par la base de données et le pilote.

40.13.8. La possibilité d'avoir plusieurs ResultSet retournés par un CallableStatement ouverts en même temps

La version 2 de l'API JDBC ne permet à un objet Statement de n'avoir qu'un seul ResultSet ouvert à un instant donné.

La version 3 de l'API propose une fonctionnalité pour outrepasser cette limitation. Par défaut, la méthode execute() ferme le ResultSet retourné par sa précédente exécution. L'interface Statement a été enrichie d'une nouvelle méthode nommée getMoreResults(). Cette méthode attend un paramètre qui peut prendre les valeurs :

CLOSE_ALL_RESULTS	Les ResultSets précédemment ouvert sont fermés à l'appel de la méthode
CLOSE_CURRENT_RESULT	L'objet ResultSet courant est fermé lors de l'appel à la méthode
KEEP_CURRENT_RESULT	L'objet ResultSet courant reste ouvert lors de l'appel à la méthode

Elle retourne un booléen qui vaut true si il y a encore au moins un ResultSet à traiter.

Cette fonctionnalité peut être pratique notamment pour utiliser des procédures stockées qui renvoie plusieurs curseurs de données.

40.13.9. Préciser si un ResultSet doit être maintenu ouvert ou fermé à la fin d'une transaction

Un Resulset est automatiquement fermé à la fin d'une transaction. JDBC 3.0 propose une fonctionnalité qui permet de préciser si dans ce cas le ResultSet doit être maintenu ouvert ou fermé.

Une version surchargé des méthodes createStatement(), prepareCall() et prepareStatement() de la classe Connection attend en paramètre un entier nommé resultSetHoldability qui peut prendre les valeurs :

ResultSet.HOLD_CURSORS_OVER_COMMIT	maintient l'objet ouvert après l'exécution d'un commit d'une transaction
ResultSet.CLOSE_CURSORS_AT_COMMIT	ferme l'objet après l'exécution d'un commit d'une transaction

40.13.10. La mise à jour des données de type BLOB, CLOB, REF et ARRAY

La norme SQL99 propose les types de données BLOB (Binary Large OBject) et CLOB (Character Large OBject) pour permettre la gestion des données de grande taille respectivement de type binaire ou chaîne de caractères.

JDBC 2.0 ne proposait que des fonctionnalités pour lire des données de ces types. Chaque pilote souhaitant proposer des fonctionnalités pour les mettre à jour le faisait de façon particulière : ceci rend le code dépendant du fournisseur du pilote.

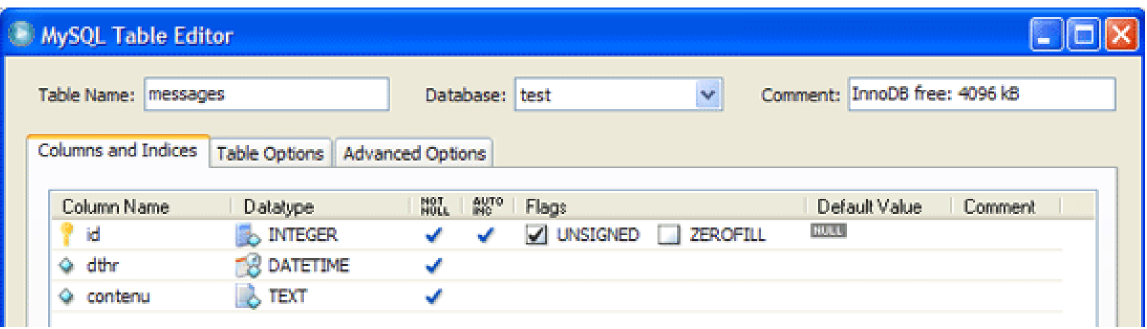
JDBC 3.0 propose en standard un mécanisme pour mettre à jour les champs de type BLOB et CLOB.

L'API propose dans l'interface java.sql.Blob une nouvelle méthode setBinaryStream() qui renvoie un objet de type OutputStream.

L'API propose dans l'interface java.sql.Clob plusieurs méthodes pour modifier le contenu du champ :

- setString() qui modifie le contenu avec la chaîne de caractères à partir de la position fournie en paramètre
- setAsciiStream() qui renvoie un objet de type Writer pour traiter un flux au format Ascii
- setCharacterStream() qui renvoie un objet de type Writer pour traiter un flux au format Unicode

L'exemple ci-dessous utilise la table suivante :



Exemple :

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.Writer;
import java.sql.Clob;
import java.sql.Connection;
import java.sql.Date;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
```

```
public class TestJdbc103 {

  public static void main(java.lang.String[] args) {
    Connection con = null;
    Statement stmt = null;
    PreparedStatement pstmt = null;
    ResultSet resultats = null;
    String requete = "";

    // chargement du pilote
    try {
      Class.forName("com.mysql.jdbc.Driver").newInstance();
    } catch (Exception e) {
      e.printStackTrace();
      System.exit(99);
    }

    try {
      String DBurl = "jdbc:mysql://localhost:3306/test";
      con = DriverManager.getConnection(DBurl);
      pstmt = con
          .prepareStatement("insert into messages (dthr, contenu)  Values (?, ?) ");

      pstmt.setDate(1, new Date(new java.util.Date().getTime()));

      Clob contenu = con.createClob();
      Writer writer = contenu.setCharacterStream(1);
      writer.write("contenu du message 1");
      writer.close();

      pstmt.setClob(2, contenu);
      pstmt.executeUpdate();

      // creation et execution de la requête
      requete = "SELECT id, dthr, contenu FROM messages where id=1";
      stmt = con.createStatement();
      resultats = stmt.executeQuery(requete);
      resultats.next();
      contenu = resultats.getClob(3);
      System.out.println("contenu=" + ClobToString(contenu));

    } catch (SQLException e) {
      e.printStackTrace();
      if (con != null) {
        try {
          con.rollback();
        } catch (SQLException ex) {
        }
      }
    } catch (IOException e) {
      e.printStackTrace();
    } finally {
      if (resultats != null) {
        try {
          resultats.close();
        } catch (SQLException ex) {
        }
      }
      if (stmt != null) {
        try {
          stmt.close();
        } catch (SQLException ex) {
        }
      }
      if (pstmt != null) {
        try {
          pstmt.close();
        } catch (SQLException ex) {
        }
      }
      if (con != null) {
        try {
          con.close();
```

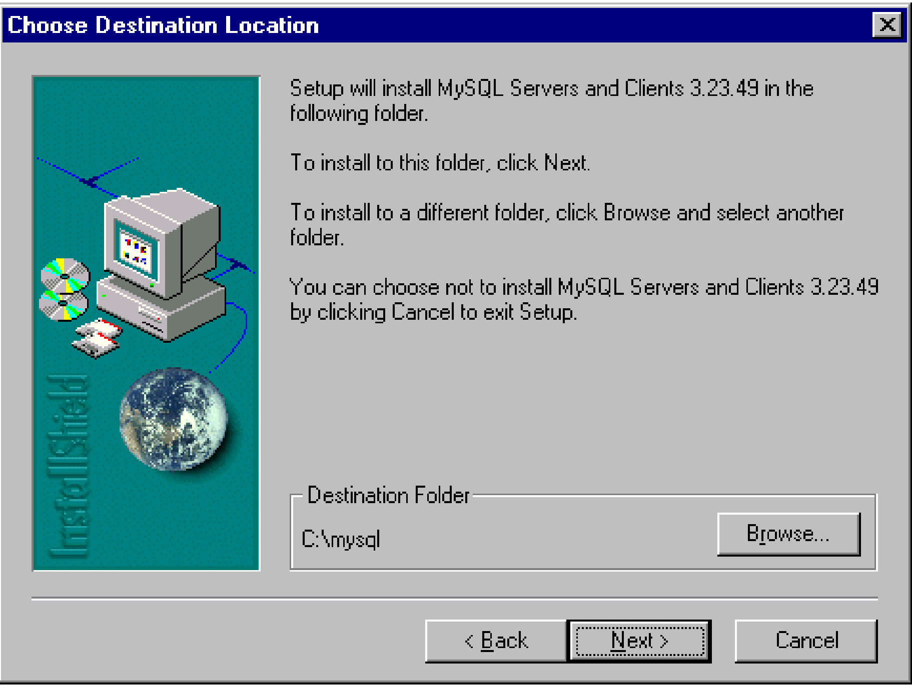
```
        } catch (SQLException ex) {
        }
      }
    }

    System.exit(0);
  }

  public static String ClobToString(Clob cl) throws IOException, SQLException {
    StringBuffer resultat = new StringBuffer("");
    if (cl != null) {
      String ligne = null;

      BufferedReader br = new BufferedReader(cl.getCharacterStream());

      while ((ligne = br.readLine()) != null)
        resultat.append(ligne);
    }
    return resultat.toString();
  }

}
```

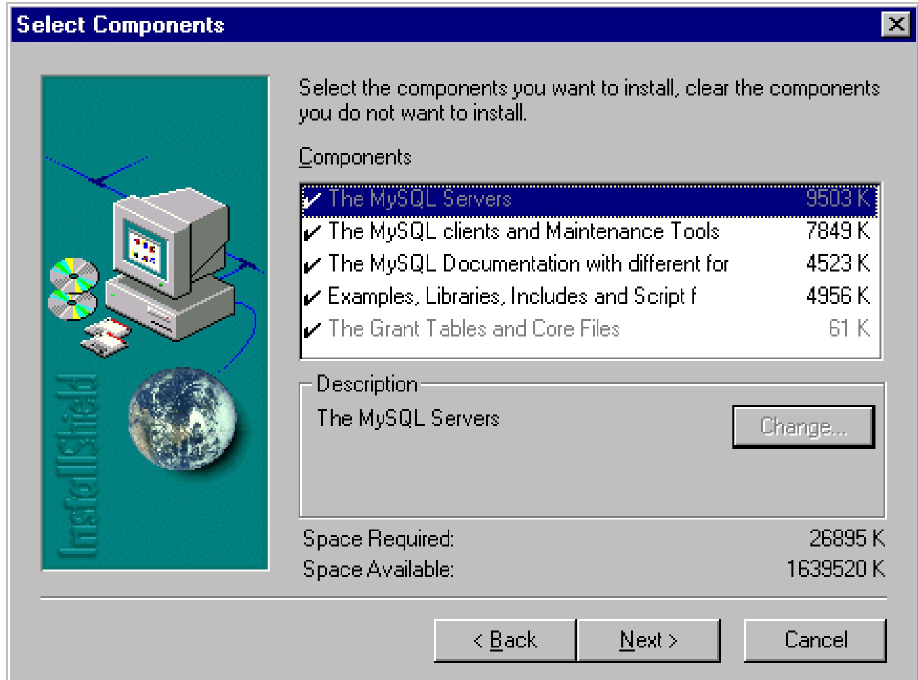
40.14. MySQL et Java

MySQL est une des bases de données open source les plus populaires.

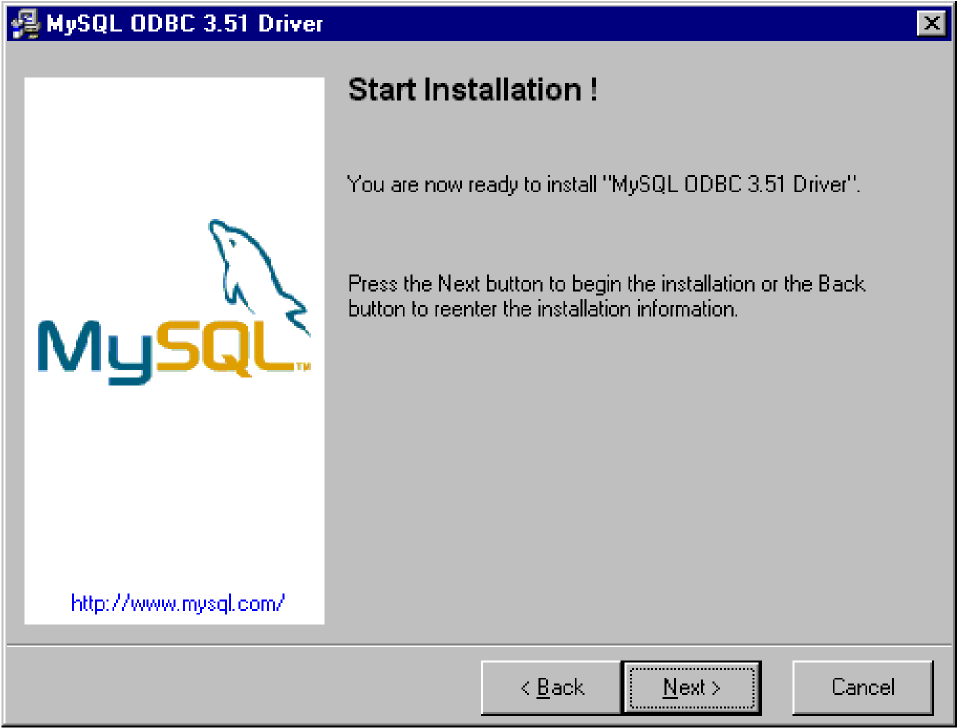
40.14.1. L'installation de MySQL 3.23 sous Windows

Il suffit de télécharger le fichier mysql-3.23.49-win.zip sur le site www.mysql.com, de décompresser ce fichier dans un répertoire et d'exécuter le fichier setup.exe





Il faut ensuite télécharger le pilote ODBC, MyODBC-3.51.03.exe, et l'exécuter



40.14.2. Les opérations de base avec MySQL

Cette section est une présentation rapide de quelques fonctionnalités de base pour pouvoir utiliser MySQL. Pour un complément d'informations sur toutes les possibilités de MySQL, consultez la documentation de cet excellent outil.

Pour utiliser MySQL, il faut s'assurer que le serveur est lancé sinon il faut exécuter la commande c:\mysql\bin\mysqld-max

Pour exécuter des commandes SQL, il faut utiliser l'outil c:\mysql\bin\mysql. Cet outil est un interpréteur de commandes en mode console.

Exemple : pour voir les databases existantes

```
mysql>show databases;
+----------+
| Database |
+----------+
| mysql    |
| test     |
+----------+
2 rows in set (0.00 sec)
```

Un des premières choses à faire, c'est de créer une base de données qui va recevoir les différentes tables.

Exemple : Pour créer une nouvelle base de données nommée "testjava"

```
mysql> create database testjava;
Query OK, 1 row affected (0.00 sec)

mysql>use testjava;
Database changed
```

Cette nouvelle base de données ne contient aucune table. Il faut créer la ou les tables utiles aux développements.

Exemple : Création d'une table nommée personne contenant trois champs : nom, prenom et date de naissance

```
mysql> show tables;
Empty set (0.06 sec)

mysql> create table personne (nom varchar(30), prenom varchar(30), datenais date
);
Query OK, 0 rows affected (0.00 sec)

mysql>show tables;
+--------------------+
| Tables_in_testjava |
+--------------------+
| personne           |
+--------------------+
1 row in set (0.00 sec)
```

Pour voir la définition d'une table, il faut utiliser la commande DESCRIBE :

Exemple : voir la définition de la table

```
mysql> describe personne;
+----------+-------------+------+-----+---------+-------+
| Field    | Type        | Null | Key | Default | Extra |
+----------+-------------+------+-----+---------+-------+
| nom      | varchar(30) | YES  |     | NULL    |       |
| prenom   | varchar(30) | YES  |     | NULL    |       |
| datenais | date        | YES  |     | NULL    |       |
+----------+-------------+------+-----+---------+-------+
3 rows in set (0.00 sec)
```

Cette table ne contient aucun enregistrement. Pour ajouter un enregistrement, il faut utiliser la commande SQL insert.

Exemple : insertion d'une ligne dans la table

```
mysql> select * from personne;
Empty set (0.00 sec)

mysql> insert into personne values ('Nom 1','Prenom 1','1970-08-11');
Query OK, 1 row affected (0.05 sec)
```

```
mysql> select * from personne;
+--------+----------+------------+
| nom    | prenom   | datenais   |
+--------+----------+------------+
| Nom 1  | Prenom 1 | 1970-08-11 |
+--------+----------+------------+
1 row in set (0.00 sec)
```

Il existe des outils graphiques libres ou commerciaux pour faciliter l'administration et l'utilisation de MySQL.

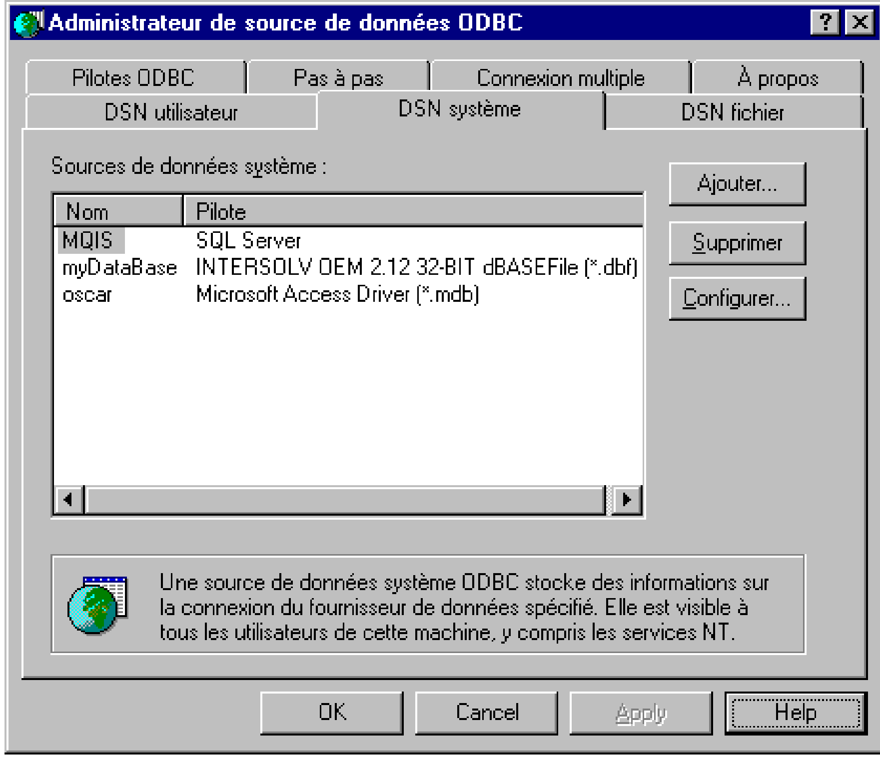
40.14.3. L'utilisation de MySQL avec Java via ODBC

Sous Windows, il est possible d'utiliser une base de données MySQL avec Java en utilisant ODBC. Dans ce cas, il faut définir une source de données ODBC sur la base de données et l'utiliser avec le pilote de type 1 fourni en standard avec J2SE.

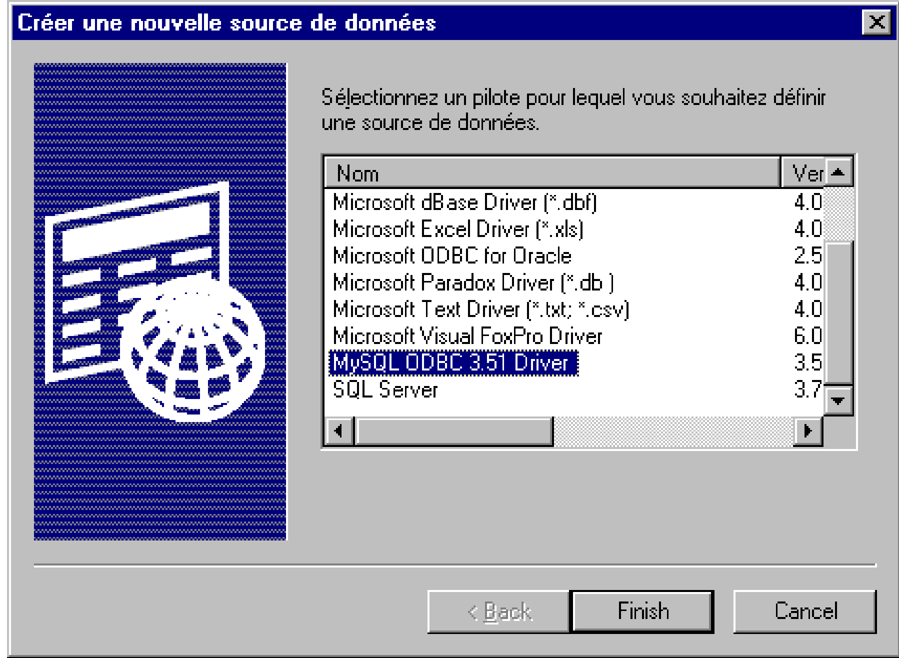
40.14.3.1. La déclaration d'une source de données ODBC vers la base de données

Dans le panneau de configuration, cliquez sur l'icône " Source de données ODBC ".

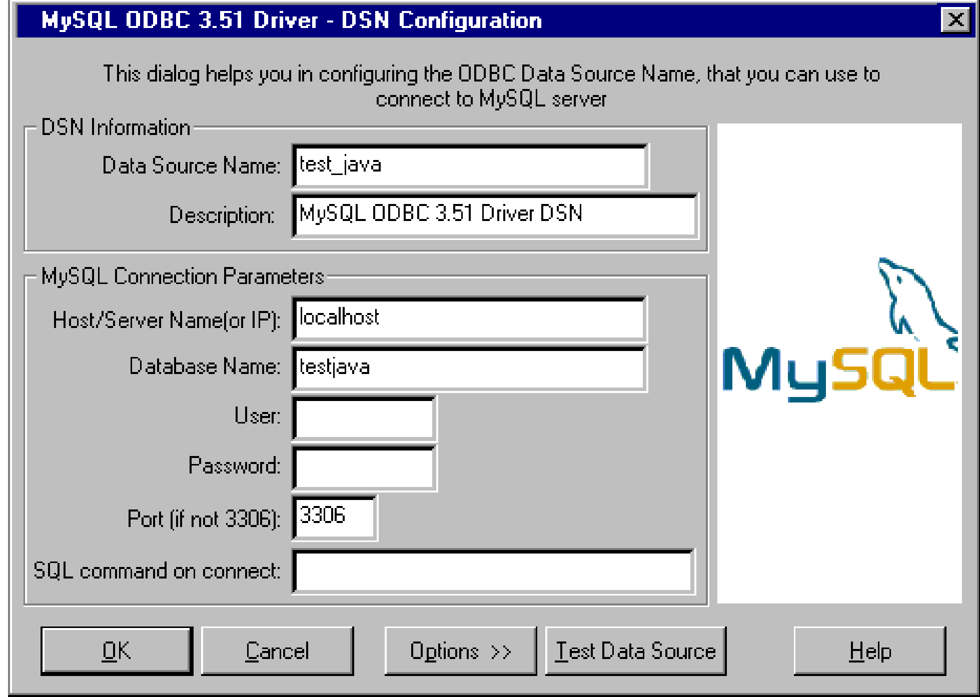
Le plus simple est de créer une source de données Systeme qui pourra être utilisée par tous les utilisateurs en cliquant sur l'onglet " DSN système "



Pour ajouter une nouvelle source de données, il suffit de cliquer sur le bouton "Ajouter ... ". Une boîte de dialogue permet de sélectionner le type de pilote qui sera utilisé par la source de données.



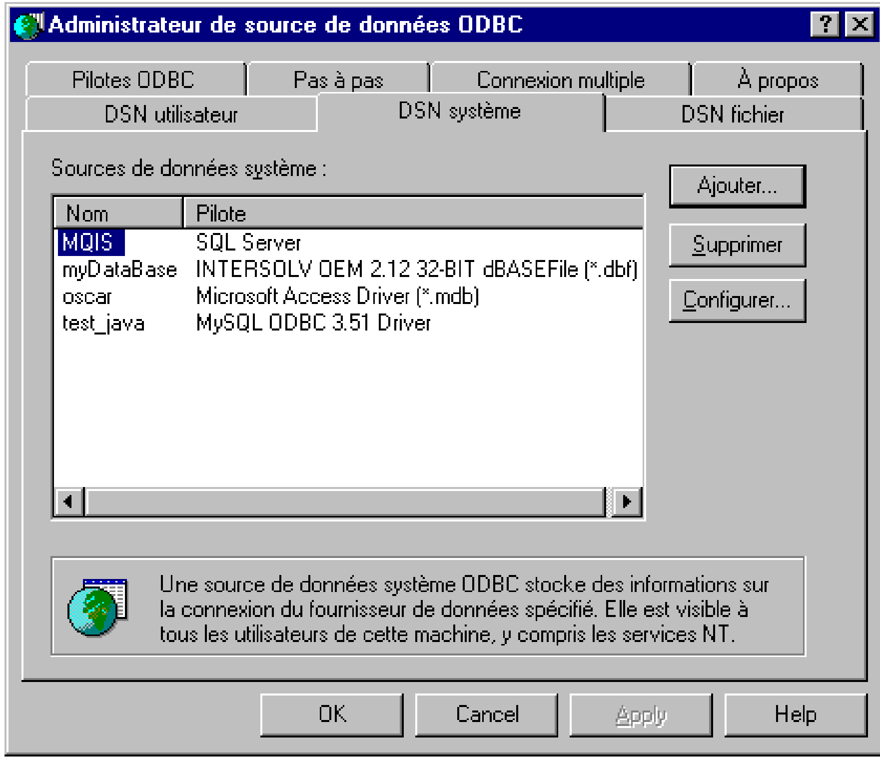
Il faut sélectionner le pilote MySQL et cliquer sur le bouton "Finish".



Une nouvelle boîte de dialogue permet de renseigner les informations sur la base de données à utiliser notamment le nom de DSN et le nom de la base de données.

Pour vérifier si la connexion est possible, il suffit de cliquer sur le bouton " Test Data Source "

Cliquer sur Ok pour fermer la fenêtre et cliquer sur Ok pour valider les paramètres et créer la source de données.



La source de données est créée.

40.14.3.2. L'utilisation de la source de données

Pour utiliser la source de données, il faut écrire et tester une classe Java. La seule particularité est l'utilisation du pont JDBC-ODBC comme pilote JDBC et l'URL spécifique à ce pilote qui contient le nom de la source de données définie.

Exemple :

```
import java.sql.*;

public class TestJDBC10 {

  private static void affiche(String message) {
    System.out.println(message);
  }

  private static void arret(String message) {
    System.err.println(message);
    System.exit(99);
  }

  public static void main(java.lang.String[] args) {
    Connection con = null;
    ResultSet resultats = null;
    String requete = "";

     // chargement du pilote
    try {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    } catch (ClassNotFoundException e) {
      arret("Impossible de charger le pilote jdbc:odbc");
    }

    //connection a la base de données
    affiche("connection a la base de donnees");
    try {

      String DBurl = "jdbc:odbc:test_java";
      con = DriverManager.getConnection(DBurl);
    } catch (SQLException e) {
```

```
      arret("Connection à la base de donnees impossible");
    }

    //creation et execution de la requête
    affiche("creation et execution de la requête");
    requete = "SELECT * FROM personne";

    try {
      Statement stmt = con.createStatement();
      resultats = stmt.executeQuery(requete);
    } catch (SQLException e) {
      arret("Anomalie lors de l'execution de la requête");
    }

    //parcours des données retournees
    affiche("parcours des données retournees");
    try {
      ResultSetMetaData rsmd = resultats.getMetaData();
      int nbCols = rsmd.getColumnCount();
      boolean encore = resultats.next();

      while (encore) {

        for (int i = 1; i <= nbCols; i++)
          System.out.print(resultats.getString(i) + "");

        System.out.println();

        encore = resultats.next();
      }

      resultats.close();
    } catch (SQLException e) {
      arret(e.getMessage());
    }

    affiche("fin du programme");
    System.exit(0);
  }
}
```

Résultat :

```
C:\$user>javac TestJDBC10.java
C:\$user>java TestJDBC10
connexion a la base de donnees
creation et execution de la requ_te
parcours des donn_es retournees
Nom 1 Prenom 1 1970-08-11
fin du programme
```

40.14.4. L'utilisation de MySQL avec Java via un pilote JDBC

mm.mysql est un pilote JDBC de type IV développé sous licence LGPL par Mark Matthews pour accéder à une base de données MySQL.

Le téléchargement du pilote JDBC se fait sur le site http://mmmysql.sourceforge.net/ . Le fichier mm.mysql-2.0.14-you-must-unjar-me.jar contient les sources et les binaires du pilote.

Pour utiliser cette archive, il faut la décompresser, par exemple dans le répertoire d'installation de mysql.

Il faut s'assurer que les fichiers jar sont accessibles dans le classpath ou les préciser manuellement lors de la compilation et de l'exécution comme dans l'exemple ci dessous.

Exemple :

```
import java.sql.*;

public class TestJDBC11 {

  private static void affiche(String message) {
    System.out.println(message);
  }

  private static void arret(String message) {
    System.err.println(message);
    System.exit(99);
  }

  public static void main(java.lang.String[] args) {
    Connection con = null;
    ResultSetresultats = null;
    String requete = "";

    // chargement du pilote
    try {
      Class.forName("org.gjt.mm.mysql.Driver").newInstance();
    } catch (Exception e) {
      arret("Impossible decharger le pilote jdbc pour mySQL");
    }

    //connexion a la base de données
    affiche("connexion a la base de donnees");
    try {

      String DBurl = "jdbc:mysql://localhost/testjava";
      con = DriverManager.getConnection(DBurl);
    } catch (SQLException e) {
      arret("Connexion a la base de donnees impossible");
    }

    //creation et execution de la requête
    affiche("creation et execution dela requête");
    requete = "SELECT * FROM personne";

    try {
      Statement stmt = con.createStatement();
      resultats = stmt.executeQuery(requete);
    } catch (SQLException e) {
      arret("Anomalie lors de l'execution de la requete");
    }

    //parcours des données retournees
    affiche("Parcours des donnees retournees");
    try {
      ResultSetMetaData rsmd = resultats.getMetaData();
      int nbCols = rsmd.getColumnCount();
      boolean encore = resultats.next();

      while (encore) {

        for (int i = 1; i <= nbCols; i++)
          System.out.print(resultats.getString(i) + "");

        System.out.println();
        encore = resultats.next();
      }

      resultats.close();
    } catch (SQLException e) {
      arret(e.getMessage());
    }

    affiche("fin du programme");
    System.exit(0);
  }
}
```

Le programme est identique au précédent utilisant ODBC sauf :

- le nom de la classe du pilote
- l'URL de connexion à la base qui dépend du pilote

Résultat :

```
C:\$user>javac -classpath c:\j2sdk1.4.0-rc\jre\lib\mm.mysql-2.0.14-bin.jar TestJDBC11.java
C:\$user>
C:\$user>java -cp .;c:\j2sdk1.4.0-rc\jre\lib\mm.mysql-2.0.14-bin.jar TestJDBC11
connexion a la base de donnees
creation et execution de la requ_te
Parcours des donnees retournees
Nom 1 Prenom 1 1970-08-11
fin du programme
```

40.15. L'amélioration des performances avec JDBC

Les opérations d'accès à une base de données sont généralement nombreuses et source de nombreux ralentissements dans une application : il est donc nécessaire de procéder à des opérations de tuning sur ces traitements.

Ces opérations doivent être prises en compte dès le début d'un projet.

Comme pour toutes opérations de tuning, un outil de test de charge et de monitoring sont nécessaires pour pouvoir quantifier les performances des accès aux bases de données.

Le choix des outils utilisés peut grandement influencer sur les performances notamment :

- La version du JRE
- Le pilote JDBC (la version de JDBC supportée, optimisations proposées, cache, ...)

Voici quelques recommandations de base qui permettent d'améliorer les performances regroupées par catégories.

40.15.1. Le choix du pilote JDBC à utiliser

La qualité du pilote JDBC est importante notamment en termes de rapidité, type de pilote, version de JDBC supportée, ...

Le type du pilote influe grandement sur les performances :

- Le type 1 (pont JDBC/ODBC) : les pilotes de ce type sont à éviter car les différentes couches mises en oeuvre (JDBC, pilote JDBC, ODBC, pilote ODBC, base de données) dégradent les performances
- Le type 2 (utilise une API native) : les pilotes de ce type ont généralement des performances moyennes
- Le type 3 (JDBC, pilote JDBC, middleware, DB) : les pilotes de type 3 communiquent avec un middleware généralement sur le serveur. Ils sont généralement plus performants que ceux de type 1 et 2
- Le type 4 (JDBC, pilote JDBC, DB) les pilotes de type 4 offre généralement les meilleures performances car ils sont écrits en Java et communiquent directement avec la base de données

Il est donc préférable d'utiliser des pilotes de type 4 ou 3.

Il peut être intéressant de tester le pilote proposé par le fournisseur de la base de données mais aussi de tester des pilotes fournis par des tiers.

Il est préférable d'utiliser un pilote qui supporte la version la plus récente de JDBC.

40.15.2. La mise en oeuvre de best practices

Plusieurs best practices sont communément mises en oeuvre lors de l'utilisation de JDBC :

- Fermer les ressources inutilisées dès que possible (Connection, Statement, ResultSet)
- Limiter le nombre de données retournées par une requête SQL uniquement à celles utiles
- Toujours assurer un traitement des warnings et des exceptions

40.15.3. L'utilisation des connexions et des Statements

Il est préférable de maintenir une connexion ouverte et la réutilisée plutôt que créer une nouvelle connexion et la fermer à chaque opération sur la base de données. C'est ce que permettent les pools de connexions.

Si les accès sont en lecture seule, il est préférable d'utiliser la méthode setReadOnly() de l'objet Connection en lui passant le paramètre true pour permettre au pilote de faire des optimisations.

Il est possible de paramétrer la quantité de données reçues de la base de données en utilisant les méthodes setMaxRows(), setMaxFieldSize() et SetFetchSize() de l'interface Statement.

La méthode nativeSQL() de la classe Connection permet d'obtenir la requête SQL native qui sera envoyée par le pilote à la base de données.

40.15.4. L'utilisation d'un pool de connexions

La création d'une connexion vers une base de données est coûteuse en temps et en ressources. Le rôle d'un pool de connexions est de maintenir un certain nombre de connexions ouvertes à disposition de l'application dans un cache et de les proposer aux besoins.

Un pool peut être fourni par l'environnement d'exécution (par exemple un serveur d'application) soit être fourni par un tiers (il en existe plusieurs en open source) soit être développé de toute pièce.

L'utilisation d'un pool de connexions est sûrement l'action la plus efficace pour des applications qui utilisent les accès à la base de données de façon importante.

Il peut être important de configurer correctement le pool de connexions utilisé notamment la taille du pool pour limiter la création et la destruction des connexions.

Un pool de connexions peut fonctionner selon deux modes principaux :

- Taille fixe : l'obtention d'une connexion alors que toutes celles du pool sont en cours d'utilisation implique l'attente de la libération d'une des connexions
- Taille variable : le pool possède une taille minimale et maximale avec une possibilité d'extension en cas de surcharge de travail

40.15.5. La configuration et l'utilisation des ResultSets en fonction des besoins

Une bonne configuration et utilisation des objets de type ResultSet peuvent améliorer les performances.

Il faut utiliser le curseur adapté aux besoins :

- TYPE_FORWARD_ONLY: aucune mise à jour, à utiliser pour des lectures séquentielles
- TYPE_SCROLL-SENSITIVE: parcours avec mise à jour immédiate
- TYPE_SCROLL_INSENSITIVE: parcours avec mises à jour à la fermeture de la connexion. Il faut éviter ce type pour des requêtes qui retournent qu'une seule occurrence.

Il faut éviter d'utiliser la méthode getObject() mais utiliser la méthode getXXX() adaptée au type d'une donnée pour extraire sa valeur.

40.15.6. L'utilisation des PreparedStatement

Il est intéressant d'utiliser les PreparedStatement notamment pour les requêtes qui sont exécutées plusieurs fois avec les mêmes paramètres ou des paramètres différents (les valeurs des données fournies à la requête peuvent être paramétrées).

Une même requête exécutée avec des paramètres différents nécessite certains traitements identiques par la base de données : une partie de ces traitements sont réalisés une et une seule fois lors de la première utilisation d'un PreparedStatement par une connexion. Les appels suivants avec la même connexion sont plus rapides puisque ces traitements ne sont pas refaits.

A partir de JDBC 3.0, les objets de type PreparedStatement peuvent être stockés dans un cache partagé des connexions d'un même pool : ceci améliore les performances car cela évite d'avoir certaines opérations mises en oeuvre à chaque appel (vérification de la syntaxe, optimisation des chemins d'accès et des plans d'exécution, ...).

40.15.7. La maximisation des traitements effectués par la base de données :

Par exemple pour obtenir un nombre d'occurrences, il est préférable d'effectuer une requête SQL contenant un count(*) plutôt que de parcourir un ResultSet avec un compteur incrémenté à chaque itération.

Il est possible d'utiliser les procédures stockées pour les traitements lourds ou complexes sur la base de données plutôt que d'effectuer plusieurs appels à la base de données pour réaliser les mêmes traitements côté Java. Les performances sont accrues car les traitements sont réalisés par la base de données ce qui évite notamment des échanges réseaux.

Attention ceci n'est vrai que pour des traitements complexes : une simple requête SQL s'exécutera plus rapidement que d'appeler une procédure stockée qui contient simplement la requête.

Il est préférable d'utiliser les marqueurs de paramètres dans les requêtes des objets de type Statement plutôt que de les passer en dur dans la requête.

40.15.8. L'exécution de plusieurs requêtes en mode batch

Il est possible d'exécuter de nombreuses requêtes en utilisant les BatchUpdates : ceci permet de regrouper plusieurs opérations sur la base de données en un seul appel.

Pour mettre en oeuvre le BatchUpdates, il faut :

- Inhiber l'autocommit en utilisant la méthode setAutoCommit(false) de l'objet Connection
- Ajouter les traitements SQL en utilisant la méthode Statement.addBtach()
- Exécuter les traitements en utilisant la méthode Statement.executeBatch()

40.15.9. Prêter une attention particulière aux transactions

Il faut minimiser les conflits engendrés par les transactions (deadlocks notamment)

Par défaut, une connexion est en mode autocommit ce qui implique la création et la validation d'une transaction à chaque opération.

L'autocommit qui est le mode par défaut pour une connexion implique une nouvelle transaction pour chaque opération réalisée.

Il est donc préférable d'inhiber l'autocommit en passant false à la méthode setAutoCommit() et de réaliser plusieurs opérations dans une même transaction avant de la valider par un commit. Il ne faut cependant pas laisser une transaction ouverte trop longtemps pour éviter des problèmes de concurrence d'accès : une transaction posant des verrous sur la base de données, il est important de minimiser le temps d'exécution d'une transaction.

Le choix du mode de transaction influe sur les performances. Il faut choisir en fonction des besoins car plus le niveau d'isolation est important moins les performances sont bonnes : TRANSACTION_NONE, TRANSACTION_READ_UNCOMMITED, TRANSACTION_READ_COMMITED, TRANSACTION_REPEATABLE_READ, TRANSACTION_SERIALIZABLE

La méthode setTransactionIsolation() permet de préciser le mode de transaction à utiliser.

Il est préférable d'éviter d'utiliser les transactions distribuées autant que possible.

40.15.10. L'utilisation des fonctionnalités de JDBC 3.0

JDBC 3.0 propose des fonctionnalités pour améliorer les performances notamment au niveau du cache des connexions et des objets de type PreparedStatement, les objets RowSet, ...

Le pool de connexion et le pool de Statement travaillent ensemble pour qu'une connexion puisse utiliser un objet Statement du pool qui a été créé par une autre connexion. Ainsi un objet de type Statement n'est plus lié à une connexion mais partagé entre les connexions d'un même pool ce qui améliore encore les performances.

Un objet de typeCacheRowSet permet d'obtenir des données, de libérer la connexion, de les modifier en local et de les resynchroniser dans la base de données avec une nouvelle connexion. Il n'est donc pas nécessaire d'avoir une connexion ouverte durant tous les traitements. Il faut cependant prêter une attention particulière aux éventuels conflits de mise à jour.

Les savePoints sont assez gourmands en ressources : il est nécessaire de libérer ces ressources en utilisant la méthode releaseSavePoint() de la classe Connection.

40.15.11. Les optimisations sur la base de données

Les optimisations côté Java sont importantes mais il est aussi nécessaire de procéder à des optimisations côté base de données, généralement réalisées par un DBA dans des structures de taille moyenne ou importante.

Les quelques optimisations fournies ci dessous sont assez généralistes : elles ne dispensent pas d'effectuer des optimisations spécifiques à la base de données utilisées.

- Il faut mettre en place les index utiles : l'ajout d'un index peut dramatiquement améliorer les performances mais trop d'index nuit car la base de données doit les maintenir à jour.
- Les bases de données fournissent des outils pour afficher le plan d'exécution d'une requête ou d'une procédure stockée pour faciliter leur optimisation (ajout d'index, modification des clauses de la requête, ...)
- Si le pilote JDBC le permet, il peut être intéressant d'ajuster la taille des paquets échangés avec la base de données

- Utiliser le type de données approprié aux données stockées en fonction des besoins (exemple : représenter une date avec un type DateTime (plus de sécurité dans l'utilisation de la donnée) ou varchar (traitement plus rapide))
- Il est préférable de stocker les chaînes de caractères en Unicode (encodage en UTF-8 par exemple) dans la base de données pour éviter les conversions. Ceci a cependant un impact important sur la taille de la base de données

40.15.12. L'utilisation d'un cache

L'utilisation d'un cache pour stocker les données peut éviter des accès à la base de données. Ceci est particulièrement adapté pour des données lues de façon répétitives ou dont les valeurs évoluent très peu ou pas du tout (données en lecture

seule, données de références, ...).

Il faut cependant faire attention à la durée de vie des objets dans le cache afin d'éviter des problèmes de rafraichissement de données.

Il ne faut pas mettre en cache les objets de types ResultSet : il faut le parcourir, stocker les données dans des objets du domaine et mettre ces objets dans le cache.

40.16. Les ressources relatives à JDBC

- La page de Sun du JDBC
  http://java.sun.com/products/jdbc/
- Le didacticiel de Sun sur JDBC
  http://java.sun.com/docs/books/tutorial/jdbc/
- La documentation du package java.sql
  http://java.sun.com/j2se/1.4/docs/api/java/sql/package-summary.html
- Liste des pilotes JDBC de Sun
  http://industry.java.sun.com/products/jdbc/drivers/

41. JDO (Java Data Object)

JDO (Java Data Object) est la spécification du JCP numéro 12 qui propose une technologie pour assurer la persistance d'objets Java dans un système de gestion de données. La spécification regroupe un ensemble d'interfaces et de règles qui doivent être implémentées par un fournisseur tiers.

La version 1.0 de cette spécification a été validée au premier trimestre 2002. Elle devrait connaître un grand succès car le mapping entre des données stockées dans un format particulier (bases de données ...) et un objet a toujours été difficile. JDO propose de faciliter cette tâche en fournissant un standard.

Ce chapitre contient plusieurs sections :

- La présentation de JDO
- Un exemple avec Lido
- L'API JDO
- La mise en oeuvre
- Le parcours de toutes les occurrences
- La mise en oeuvre de requêtes

41.1. La présentation de JDO

Les principaux buts de JDO sont :

- la facilité d'utilisation (gestion automatique du mapping des données)
- la persistance universelle : persistance vers tout type de système de gestion de ressources (bases de données relationnelles, fichiers, ...)
- la transparence vis à vis du système de gestion de ressources utilisé : ce n'est plus le développeur mais JDO qui dialogue avec le système de gestion de ressources
- la standardisation des accès aux données
- la prise en compte des transactions

Le développement avec JDO se déroule en plusieurs étapes :

1. écriture des objets contenant les données (des beans qui encapsulent les données) : un tel objet est nommé instance JDO
2. écriture des objets qui utilisent les objets métiers pour répondre aux besoins fonctionnels. Ces objets utilisent l'API JDO.
3. écriture du fichier metadata qui précise le mapping entre les objets et le système de gestion des ressources. Cette partie est très dépendante du système de gestion de ressources utilisé
4. enrichissement des objets métiers
5. configuration du système de gestion des ressources

JDBC et JDO ont les différences suivantes :

JDBC	JDO
orienté SQL	orienté objets

le code doit être ajouté explicitement	code est ajouté automatiquement
	gestion d'un cache
	mapping réalisé automatiquement ou à l'aide d'un fichier de configuration au format XML
utilisation avec un SGBD uniquement	utilisation de tout type de format de stockage

JDO est une spécification qui définit un standard : pour pouvoir l'utiliser il faut utiliser une implémentation fournie par un fournisseur. Plusieurs implémentations existent et le choix de l'une d'elle doit tenir compte des performances, du prix, du support des cibles de stockage des données, etc ... L'intérêt des spécifications est qu'il est possible d'utiliser le même code avec des implémentations différentes tant que l'on utilise uniquement les fonctionnalités précisées dans les spécifications.

Chaque implémentation est capable d'utiliser un ou plusieurs systèmes de stockage de données particulier (base de données relationnel, base de données objets, fichiers, ...).

Attention : tous les objets ne peuvent pas être rendu persistants avec JDO.

41.2. Un exemple avec Lido

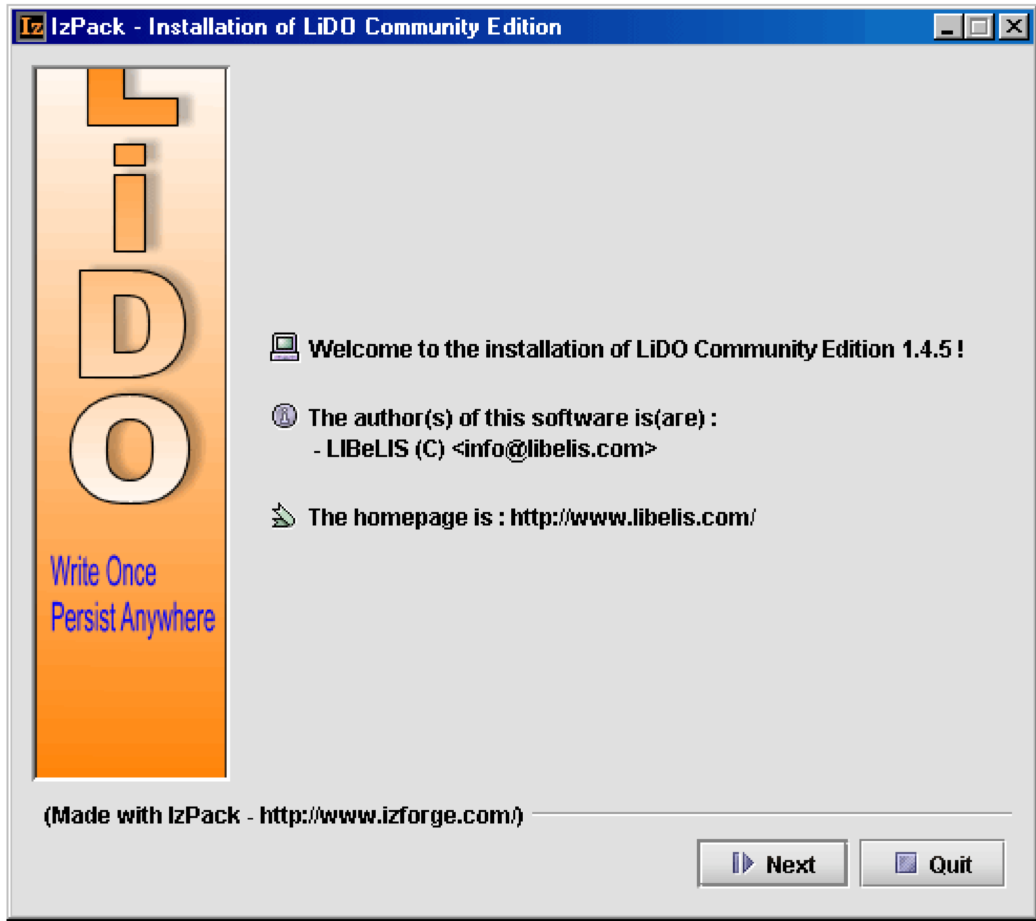
Les exemples de cette section ont été réalisés avec Lido Community Edition version 1.4.5. de la société Libelis.

Cette version est librement téléchargeable après enregistrement à l'url : http://www.libelis.com

Pour lancer l'installation, il suffit de double cliquer sur le fichier LiDO_Community_1[1].4.5.jar ou de saisir la commande :

Installation de Lido community edition de Libelis

```
java -jar LiDO_Community_1[1].4.5.jar
```



L'installation s'opère simplement en suivant les différentes étapes de l'assistant.

Le premier exemple permet simplement de rendre persistant un objet instancié dans une base de données MySQL.

Pour faciliter la mise en oeuvre des différentes étapes, un script batch pour Windows sera écrit tout au long de cette section et exécuté. Ce script débute par une initialisation de certaines variables d'environnement.

Début du script

```
@echo off
REM - script permettant la compilation, l'enrichissement, la creation du schema de
REM - base de données et l'execution du code de test de JDO avec Lido 1.4.5.
set LIDO_HOME=C:\java\LiDO
set JAVA_HOME=C:\java\j2sdk1.4.2_02

echo initialisation
echo.
SET PATH=%LIDO_HOME%\bin;%JAVA_HOME%\bin

SET CLASSPATH=.;.\mm.mysql-2.0.14-bin.jar
SET CLASSPATH=%CLASSPATH%;%JAVA_HOME%\lib\tools.jar
SET CLASSPATH=%CLASSPATH%;%LIDO_HOME%\lib\lido-api.jar
SET CLASSPATH=%CLASSPATH%;%LIDO_HOME%\lib\jdo_1_0_0.jar
SET CLASSPATH=%CLASSPATH%;%LIDO_HOME%\lib\j2ee.jar
SET CLASSPATH=%CLASSPATH%;%LIDO_HOME%\bin
SET CLASSPATH=%CLASSPATH%;%LIDO_HOME%\lib\lido-dev.jar
SET CLASSPATH=%CLASSPATH%;%LIDO_HOME%\lib\lido-rdb.jar
SET CLASSPATH=%CLASSPATH%;%LIDO_HOME%\lib\lido-rt.jar
SET CLASSPATH=%CLASSPATH%;%LIDO_HOME%\lib\lido.tasks
SET CLASSPATH=%CLASSPATH%;%LIDO_HOME%\lib\lido.tld
SET CLASSPATH=%CLASSPATH%;%LIDO_HOME%\lib\skinlf.jar
SET CLASSPATH=%CLASSPATH%;%LIDO_HOME%\lib\connector_1_0_0.jar
SET CLASSPATH=%CLASSPATH%;%LIDO_HOME%\lib\jta_1_0_1.jar
```

Le début du script initialise 4 variables :

* LIDO_HOME : cette variable contient le chemin du répertoire dans lequel Lido a été installé
* JAVA_HOME : cette variable contient le chemin du répertoire dans lequel J2SDK a été installé
* PATH : cette variable contient les différents répertoires contenant des exécutables
* CLASSPATH : cette variable contient les différents répertoires et fichiers jar nécessaires pour la compilation et l'exécution

Pour des raisons de facilité, le répertoire courant "." est ajouté dans le CLASSPATH. Le pilote JDBC pour MySQL est aussi ajouté à cette variable.

41.2.1. La création de la classe qui va encapsuler les données

Le code de cet objet reste très simple puisque c'est simplement un bean encapsulant une personne contenant des attributs nom, prenom et datenaiss.

Exemple :

```
package testjdo;

import java.util.*;

public class Personne {
      private String nom ="";
      private String prenom = "";
      private Date datenaiss = null;

      public Personne(String pNom, String pPrenom,Date pDatenaiss) {
            nom=pNom;
            prenom=pPrenom;
            datenaiss=pDatenaiss;
      }

      public String getNom() { return nom; }

      public String getPrenom() { return prenom;}

      public Date getDatenaisse() { return datenaiss; }

      public void setNom(String pNom) { nom = pNom; }

      public void setPrenom(String pPrenom) { nom = pPrenom; }

      public void setDatenaiss(Date pDatenaiss) { datenaiss = pDatenaiss; }

}
```

41.2.2. La création de l'objet qui va assurer les actions sur les données

Cet objet va utiliser des objets JDO pour réaliser les actions sur les données. Dans l'exemple ci dessous, une seule action est codée : l'enregistrement dans la table des données du nouvel objet de type Personne instancié.

Exemple :

```
package testjdo;

import javax.jdo.*;
import java.util.*;

public class PersonnePersist {

      private PersistenceManagerFactory pmf = null;
      private PersistenceManager        pm = null;
      private Transaction               tx  = null;

      public PersonnePersist() {
            try {
```

```
            pmf = (PersistenceManagerFactory) (
            Class.forName("com.libelis.lido.PersistenceManagerFactory").newInstance());
            pmf.setConnectionDriverName("org.gjt.mm.mysql.Driver");
            pmf.setConnectionURL("jdbc:mysql://localhost/testjdo");
      } catch (Exception e ){
            e.printStackTrace();
            }
      }

      public void enregistrer() {
            Personne p = new Personne("mon nom","mon prenom",new Date());
            pm = pmf.getPersistenceManager();
            tx = pm.currentTransaction();
            tx.begin();
            pm.makePersistent(p);
            tx.commit();
            pm.close();
      }

      public static void main(String args[]) {
            PersonnePersist pp = new PersonnePersist();
            pp.enregistrer();
      }
}
```

41.2.3. La compilation

Les deux classes définies ci dessus doivent être compilées normalement en utilisant l'outil javac.

Exemple :

```
...
echo Compilation en cours
javac -classpath %CLASSPATH% testjdo\*.java
echo Compilation effectuee
echo.
...
```

41.2.4. La définition d'un fichier metadata

Le fichier metadata est un fichier au format XML qui précise le mapping à réaliser.

Exemple : metadata.jdo

```
<? xml version="1.0" ?>
<!DOCTYPE jdo SYSTEM "jdo.dtd">
<jdo>
  <package name="testjdo">
    <class name="Personne" identity-type="datastore">
      <field name="nom" />
      <field name="prenom" />
      <field name="datenaiss" />
    </class>
  </package>
</jdo>
```

41.2.5. L'enrichissement des classes contenant des données

Pour assurer une bonne exécution, il faut enrichir l'objet Personne compilé avec du code pour assurer la persistance par JDO. Lido fourni un outil pour réaliser cette tâche. Cet outil est complètement dépend de l'implémentation qui en est faite par le fournisseur de la solution JDO.

La suite du script : enrichissement

```
...
echo Enrichissement
java -cp %CLASSPATH% com.libelis.lido.Enhance -metadata metadata.jdo -verbose
echo Enrichissement effectue
echo.
...
```

Le fichier Personne.class est enrichi (sa taille passe de 867 octets à 9693 octets)

41.2.6. La définition du schéma de la base de données

Lido fournit un outil qui permet de générer les tables de la base de données contenant les tables pour le mapping des données plus des tables "techniques" nécessaires aux traitements.

Les paramètres nécessaires à l'outil de Libelis pour définir le schéma de la base de données doivent être rassemblés dans un fichier .properties.

propriété pour une base de données de type MySQL

```
# lido.properties file
# jdo standard properties
javax.jdo.option.connectionURL=jdbc:mysql://localhost/testjdo
javax.jdo.option.ConnectionDriverName=org.gjt.mm.mysql.Driver
javax.jdo.option.connectionUserName=root
javax.jdo.option.connectionPassword=
javax.jdo.option.msWait=5
javax.jdo.option.multithreaded=false
javax.jdo.option.optimistic=false
javax.jdo.option.retainValues=false
javax.jdo.option.restoreValues=true
javax.jdo.option.nontransactionalRead=true
javax.jdo.option.nontransactionalWrite=false
javax.jdo.option.ignoreCache=false


# set to PM, CACHE, or SQL to have some traces
# ex:
#lido.trace=SQL,DUMP,CACHE

#  set the Statement pool size
lido.sql.poolsize=10
lido.cache.entry-type=weak

# set the max batched statement
# 0: no batch
# default is 20

lido.sql.maxbatch=30
lido.objectpool=90

# set for PersistenceManagerFactory pool limits
lido.minPool=1
lido.maxPool=10

jdo.metadata=metadata.jdo
```

Il suffit alors d'utiliser l'application DefineSchema fournie par Lido en lui passant en paramètre le fichier .properties et le fichier .jdo

La suite du script : création du schéma de la base de données

```
 ...
echo DefineSchema en cours
java com.libelis.lido.DefineSchema -properties testjdo.properties -metadata metadata.jdo
echo DefineSchema termine
echo.
...
```

Il est facile de vérifier les traitements effectués par l'outil DefineSchema :

Exemple :

```
C:\java\testjdo>mysql
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 25 to server version: 4.0.16-nt

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> use testjdo
Database changed
mysql> show tables;
+--------------------+
| Tables_in_testjdo |
+--------------------+
| lidoidmax          |
| lidoidtable        |
| t_personne         |
+--------------------+
3 rows in set (0.00 sec)

mysql> describe lidoidmax;
+----------+------------+------+-----+---------+-------+
| Field    | Type       | Null | Key | Default | Extra |
+----------+------------+------+-----+---------+-------+
| LIDOLAST | bigint(20) | YES  |     | NULL    |       |
+----------+------------+------+-----+---------+-------+
1 row in set (0.00 sec)

mysql> describe lidoidtable;
+----------+--------------+------+-----+---------+-------+
| Field    | Type         | Null | Key | Default | Extra |
+----------+--------------+------+-----+---------+-------+
| LIDOID   | bigint(20)   |      | PRI | 0       |       |
| LIDOTYPE | varchar(255) | YES  | MUL | NULL    |       |
+----------+--------------+------+-----+---------+-------+
2 rows in set (0.00 sec)

mysql> describe t_personne;
+-----------+-------------+------+-----+---------+-------+
| Field     | Type        | Null | Key | Default | Extra |
+-----------+-------------+------+-----+---------+-------+
| LIDOID    | bigint(20)  |      | PRI | 0       |       |
| nom       | varchar(50) | YES  |     | NULL    |       |
| prenom    | varchar(50) | YES  |     | NULL    |       |
| datenaiss | datetime    | YES  |     | NULL    |       |
+-----------+-------------+------+-----+---------+-------+
4 rows in set (0.00 sec)

mysql> select * from t_personne;
Empty set (0.39 sec)
```

41.2.7. L'exécution de l'exemple

Exemple :

```
@echo off
REM - script permettant la compilation, l'enrichissement, la creation du schema de
REM - base de données et l'execution du code de test de JDO avec Libelis 1.4.5.
set LIDO_HOME=C:\java\LiDO
set JAVA_HOME=C:\java\j2sdk1.4.2_02

echo initialisation
echo.
SET PATH=%LIDO_HOME%\bin;%JAVA_HOME%\bin

SET CLASSPATH=.;.\mm.mysql-2.0.14-bin.jar
SET CLASSPATH=%CLASSPATH%;%JAVA_HOME%\lib\tools.jar
SET CLASSPATH=%CLASSPATH%;%LIDO_HOME%\lib\lido-api.jar
SET CLASSPATH=%CLASSPATH%;%LIDO_HOME%\lib\jdo_1_0_0.jar
SET CLASSPATH=%CLASSPATH%;%LIDO_HOME%\lib\j2ee.jar
SET CLASSPATH=%CLASSPATH%;%LIDO_HOME%\bin
SET CLASSPATH=%CLASSPATH%;%LIDO_HOME%\lib\lido-dev.jar
SET CLASSPATH=%CLASSPATH%;%LIDO_HOME%\lib\lido-rdb.jar
SET CLASSPATH=%CLASSPATH%;%LIDO_HOME%\lib\lido-rt.jar
SET CLASSPATH=%CLASSPATH%;%LIDO_HOME%\lib\lido.tasks
SET CLASSPATH=%CLASSPATH%;%LIDO_HOME%\lib\lido.tld
SET CLASSPATH=%CLASSPATH%;%LIDO_HOME%\lib\skinlf.jar
SET CLASSPATH=%CLASSPATH%;%LIDO_HOME%\lib\connector_1_0_0.jar
SET CLASSPATH=%CLASSPATH%;%LIDO_HOME%\lib\jta_1_0_1.jar

echo Compilation en cours
javac -classpath %CLASSPATH% testjdo\*.java
echo Compilation effectuee
echo.

echo Enrichissement
java -cp %CLASSPATH% com.libelis.lido.Enhance -metadata metadata.jdo -verbose
echo Enrichissement effectue
echo.

echo DefineSchema en cours
java com.libelis.lido.DefineSchema -properties testjdo.properties -metadata metadata.jdo
echo DefineSchema termine
echo.

echo Execution du test
java -cp %CLASSPATH% testjdo.PersonnePersist
echo Execution terminee
echo.
```

A l'issu de l'exécution, un enregistrement est créé dans la table qui mappe l'objet Personne.

Exemple :

```
 mysql> select * from t_personne;
+--------+---------+------------+---------------------+
| LIDOID | nom     | prenom     | datenaiss           |
+--------+---------+------------+---------------------+
|      1 | mon nom | mon prenom | 2004-01-23 20:06:11 |
+--------+---------+------------+---------------------+
1 row in set (0.00 sec)
```

41.3. L'API JDO

L'API de JDO se compose de deux packages :

* javax.jdo : ce package est à utiliser par les développeurs pour utiliser JDO
* javax.jdo.spi : ce package est à utiliser par les tiers pour développer une implémentation de JDO

Le package javax.jdo contient essentiellement des interfaces ainsi que quelques classes notamment la classe JDOHelper et les diverses exceptions utilisées par JDO. Les interfaces définies sont : Extent, PersistenceManager, PersistenceManagerFactory, Query etTransaction.

Les exceptions définies par l'API JDO sont : JDOCanRetryException, JDODataStoreException, JDOException, JDOFatalDataStoreException, JDOFatalException, JDOFatalInternalException, JDOFatalUserException, JDOUnsupportedOptionException et JDOUserException.

41.3.1. L'interface PersistenceManager

Cette interface définit les méthodes pour l'objet principal de l'API JDO pour les développeurs.

Certaines méthodes permettent de gérer le cycle de vie d'une instance d'un objet de type PersistenceCapable.

void close()	fermer
Transaction currentTransaction()	renvoie la transaction courante
void deletePersistent()	permet de détruire dans la source de données l'instance encapsulée
Extent getExtent(Class, boolean)	renvoie une collection d'instance encapsulant les données dans la source de données
void makePersistent()	permet de rendre persistante les données encapsulées dans l'instance en créant une nouvelle occurrence dans le système de gestion de ressources
void evict()	permet de préciser que l'instance n'est plus utilisée
Query newQuery()	renvoie un objet de type Query qui permet d'effectuer des sélections dans la source de données
void refresh()	permet de redonner à une instance les valeurs contenues dans le système de gestion de ressources

Cette interface propose aussi deux méthodes possédant de nombreuses surcharges de la méthode newQuery() pour obtenir une instance d'un objet de type Query.

41.3.2. L'interface PersistenceManagerFactory

Un objet qui implémente cette interface à pour but de fournir une instance d'une classe qui implémente l'interface PersistenceManager. Un tel objet doit être configuré via des propriétés pour instancier un objet de type PersistenceManager. Ces propriétés doivent être fournies à la fabrique avant l'instanciation du premier objet de type PersistenceManager. Il n'est dès lors plus possible de changer la configuration de la fabrique.

Cette interface possède une méthode nommée getPersistenceManager() qui permet d'obtenir une instance de la classe PersistenceManager.

41.3.3. L'interface PersistenceCapable

Cette interface doit être implémentée lors de son enrichissement par la classe qui va contenir des données. Cette classe avant son enrichissement ne doit pas implémenter cette interface : c'est lors de cette phase d'enrichissement que la classe implémentera cette interface et que seront définies les méthodes déclarées par cette interface nécessaires à la persistance des données. Cet enrichissement peut se faire de deux façons :

* manuellement : ce qui peut être long et fastidieux
* automatiquement avec un outil fourni avec l'implémentation de JDO : généralement cet outil utilise un fichier au format XML pour obtenir les informations nécessaires à la génération des méthodes

Les méthodes définies dans cette interface sont à l'usage de JDO : une fois la classe enrichie, il ne faut surtout pas appeler directement ces méthodes : elles sont toutes préfixées par jdo.

Une classe qui implémente l'interface PersistenceCapable est nommée instance JDO.

41.3.4. L'interface Query

Cette interface définit des méthodes qui permettent d'obtenir des instances représentant des données issues de la source de données.

L'interface définit plusieurs surcharges de la méthode execute() pour exécuter la requête et renvoyer un ensemble d'instances.

La méthode compile() permet de vérifier la requête et préparer son exécution.

La méthode setFilter() permet de préciser un filtre pour la requête.

Une instance d'un objet implémentant l'interface Query est obtenue en utilisant une des nombreuses surcharges de la méthode newQuery() d'un objet de type PersistenceManager.

41.3.5. L'interface Transaction

Cette interface définit les méthodes pour la gestion des transactions avec JDO.

Elle possède trois méthodes principales qui sont classiques dans la gestion des transactions :

* begin : indique le début d'une transaction
* commit : valide la transaction
* roolback : invalide la transaction et annule toutes les opérations qu'elle contient

41.3.6. L'interface Extent

Une classe qui implémente cette interface permet d'encapsuler toute une collection contenant tous les objets d'un type PersistenceCapable particulier. La méthode iterator() renvoie un objet de type Iterator qui permet de parcourir l'ensemble des éléments de la collection.

L'interface Extent ne prévoit actuellement aucun moyen de filter les éléments de la collection et il est uniquement possible d'obtenir toutes les occurrences.

La méthode close(Iterator) permet de fermer l'objet de type Iterator passé en paramètre.

41.3.7. La classe JDOHelper

La classe JDOHelper permet de faciliter l'utilisation de JDO grâce à plusieurs méthodes statiques pouvant être regroupées dans plusieurs catégories :

* connaitre l'état d'une instance JDO.

Nom	Rôle
boolean isDeleted(Object)	renvoie un booléen qui précise si l'instance JDO fournie en paramètre vient d'être supprimée dans le système de gestion de ressources
boolean isDirty(Object)	renvoie un booléen qui précise si l'instance JDO a été modifiée dans la transaction courante
boolean isNew(Object)	renvoie un booléen qui précise si l'instance JDO fournie en paramètre vient d'être rendue persistante en créant une nouvelle instance dans le système de gestion de ressources
boolean isPersistent(Object)	
boolean isTransactional(Object)	

- obtenir des objets de l'implémentation JDO

Nom	Rôle
PersistenceManager getPersistenceManager(Object)	renvoie l'objet de type PersistenceManager utilisé pour rendre persistante l'instance JDO fournie en paramètre
getObjectId(Object)	
makeDirty(Object, String)	
PersistenceManagerFactory getPersistenceManagerFactory(Properties)	

41.4. La mise en oeuvre

La mise en oeuvre de JDO requiert plusieurs étapes :

- définition d'une classe qui va encapsuler les données (instance JDO)
- définition d'une classe qui va utiliser les données
- compilation des deux classes
- définition d'un fichier de description
- enrichissement de la classe qui va contenir les données

41.4.1. La définition d'une classe qui va encapsuler les données

Une telle classe se présente sous la forme d'un bean : elle représente une occurrence particulière dans le système de stockage des données.

Cette classe n'a pas besoin ni d'utiliser ni d'importer de classes de l'API JDO.

Pour la classe qui va contenir des données, JDO impose la présence d'un constructeur sans argument. Celui ci est automatiquement ajouté à la compilation si aucun autre constructeur n'est défini, sinon il faut ajouter un constructeur sans argument manuellement.

41.4.2. La définition d'une classe qui va utiliser les données

Cette classe va réaliser des traitements en utilisant JDO pour accéder et/ou mettre à jour des données.

La suite de cette section sera développée dans une version future de ce document

41.4.3. La compilation des classes

Toutes les classes écrites doivent être compilées normalement comme toutes classes Java.

41.4.4. La définition d'un fichier de description

Pour indiquer à JDO quelles classes doivent être persistantes et préciser des informations concernant ces dernières, il faut utiliser un fichier particulier au format XML. Ce fichier désigné par "Metadata" dans les spécifications doit avoir pour extension .jdo.

Il est possible de définir un fichier de description pour chaque classes persistantes ou un fichier pour un package concernant toutes les classes persistantes du package. Dans le premier cas, le fichier doit se nommer nom_de_la_classe.jdo, dans le second nom_du_package.jdo.

Le fichier commence par un prologue :

<?xml version="1.0" encoding="UTF-8" ?>

Le fichier contient ensuite la DTD utilisée pour valider le fichier : soit une URL pointant sur la DTD du site de Sun soit une DTD sur le système de fichier.

<!DOCTYPE jdo PUBLIC "-//Sun Microsystems, Inc.//DTD Java Data Objects Metadata 1.0//EN" "http://java.sun.com/dtd/jdo_1_0.dtd">

Le tag racine du document XML est <jdo>. Ce tag peut contenir un ou plusieurs tags <package> selon les besoins, chaque tag package concernant un seul package.

Chaque tag <package> contient autant de tags <class> que de classes de type instance JDO utilisées. L'attribut "name", obligatoire, permet de préciser le nom de la classe.

Les tags <jdo>, <package>, <class> et <field> peuvent aussi avoir un tag <extension> qui va contenir des paramètres particuliers dédiés à l'implémentation de JDO utilisée. Il faut un tag <extension> pour chaque implémentation utilisée.

41.4.5. L'enrichissement de la classe qui va contenir les données

Cette phase permet d'ajouter du code à chaque classe encapsulant une instance JDO. Ce code contient les méthodes définies par l'interface PersistenceCapable.

Le ou les outils fournis par le fournisseur sont particuliers pour chaque implémentation utilisée.

41.5. Le parcours de toutes les occurrences

Un objet qui implémente l'interface Extent permet d'accéder à toutes les instances d'une classe encapsulant des données.

Un objet de type Extent est obtenu en appelant la méthode getExtent() d'un objet PersistentManager. Cette méthode attend deux paramètres : un objet de type Class qui est la classe encapsulant les données et un booléen qui permet de préciser si les sous classes doivent être prises en compte.

Un objet de type Extent ne permet qu'une seule opération sur l'ensemble des instances qu'il contient : obtenir un objet de type Iterator qui permet le parcours séquentiel de toutes les occurrences. La méthode iterator() permet de renvoyer cet objet de type Iterator : les méthodes hasNext() et next() assurent le parcours des occurrences.

L'appel de la méthode close() une fois que l'objet de type Iterator fourni en paramètre n'a plus d'utilité est obligatoire pour permettre de libérer les ressources allouées par l'objet pour son fonctionnement. La méthode closeAll() permet de fermer tout les objets de type Iterator instanciés par l'objet de type Extent.

Exemple : Afficher tous les données de la table personne

```
package testjdo;

import javax.jdo.*;
import java.util.*;

public class PersonneExtent {

  private PersistenceManagerFactory pmf = null;
  private PersistenceManager pm = null;

  public PersonneExtent() {
    try {

      pmf =
        (PersistenceManagerFactory) (Class
          .forName("com.libelis.lido.PersistenceManagerFactory")
          .newInstance());
      pmf.setConnectionDriverName("org.gjt.mm.mysql.Driver");
      pmf.setConnectionURL("jdbc:mysql://localhost/testjdo");
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  public void afficherTous() {
    pm = pmf.getPersistenceManager();

    Extent personneExtent = pm.getExtent(Personne.class, true);

    Iterator iter = personneExtent.iterator();
    while (iter.hasNext()) {
      Personne personne = (Personne) iter.next();
      System.out.println(personne.getNom() + " " + personne.getPrenom());
    }
    personneExtent.close(iter);
  }

  public static void main(String args[]) {
    PersonneExtent pe = new PersonneExtent();
    pe.afficherTous();
  }
}
```

Exemple : Contenu de la table au moment de l'exécution

```
C:\mysql\bin>mysql testjdo
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 3 to server version: 4.0.16-nt
```

```
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql< select * from t_personne;
+--------+---------+------------+---------------------+
| LIDOID | nom     | prenom     | datenaiss           |
+--------+---------+------------+---------------------+
|      1 | mon nom | mon prenom | 2004-01-23 20:06:11 |
|   1025 | Nom1    | Jean       | 2004-02-10 00:20:39 |
|   2049 | Nom4    | Jean       | 2004-02-10 00:25:09 |
|   3073 | Nom3    | Louis      | 2004-02-10 21:36:32 |
+--------+---------+------------+---------------------+
4 rows in set (0.00 sec)

mysql<
```

Résultat :

```
Execution du test
mon nom mon prenom
Nom1 Jean
Nom4 Jean
Nom3 Louis
Execution terminee
```

41.6. La mise en oeuvre de requêtes

Avec JDO, les requêtes sont mises en oeuvre grâce à un objet de type Query. Les requêtes appliquent un filtre sur un ensemble d'objets encapsulant des données sous la forme d'un objet de type Extent ou d'une collection.

Un filtre est une expression booléenne appliquée à chacune des occurrences : la requête renvoie toutes les occurrences pour laquelle le résultat de l'évaluation de l'expression est vrai. Les expressions sont exprimées avec un langage particulier nommé JDO Query Langage (JDOQL)

Une instance d'un objet qui implémente l'interface Query est obtenu en utilisant la méthode newQuery() d'un objet de type PersistanceManager.

Exemple : afficher les occurrences dont le prénom est Jean

```
package testjdo;

import javax.jdo.*;
import java.util.*;

public class PersonneQuery {

  private PersistenceManagerFactory pmf = null;
  private PersistenceManager pm = null;

  public PersonneQuery() {
    try {

      pmf =
        (PersistenceManagerFactory) (Class
          .forName("com.libelis.lido.PersistenceManagerFactory")
          .newInstance());
      pmf.setConnectionDriverName("org.gjt.mm.mysql.Driver");
      pmf.setConnectionURL("jdbc:mysql://localhost/testjdo");
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  public void filtrer() {
    pm = pmf.getPersistenceManager();

    Extent personneExtent = pm.getExtent(Personne.class, true);
```

```
    String filtre = "prenom == \"Jean\"";

    Query query = pm.newQuery(personneExtent, filtre);
    query.setOrdering("nom ascending, prenom ascending");
    Collection result = (Collection) query.execute();

    Iterator iter = result.iterator();
    while (iter.hasNext()) {
      Personne personne = (Personne) iter.next();
      System.out.println(personne.getNom() + " " + personne.getPrenom());
    }
    query.close(result);
  }

  public static void main(String args[]) {
    PersonneQuery pq = new PersonneQuery();
    pq.filtrer();
  }
}
```

Résultat :

```
Execution du test
Nom1 Jean
Nom4 Jean
Execution terminee
```

La suite de ce chapitre sera développée dans une version future de ce document

42. Hibernate

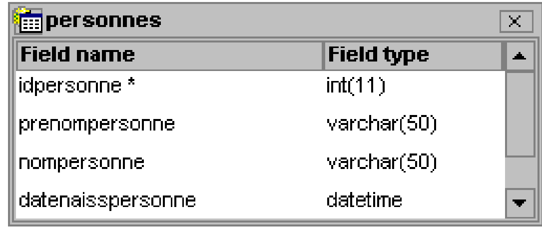
Hibernate est un projet open source visant à proposer un outil de mapping entre les objets et des données stockées dans une base de données relationnelle. Ce projet ne repose sur aucun standard mais il est très populaire notamment à cause de ses bonnes performances et de son ouverture avec de nombreuses bases de données.

Les bases de données supportées sont les principales du marché : DB2, Oracle, MySQL, PostgreSQL, Sybase, SQL Server, Sap DB, Interbase, ...

Le site officiel http://www.hibernate.org contient beaucoup d'informations sur l'outil et propose de le télécharger ainsi que sa documentation.

La version utilisée dans cette section est la 2.1.2 : il faut donc télécharger le fichier hibernate-2.1.2.zip et le décompresser dans un répertoire du système.

Cette section va utiliser Hibernate avec une base de données de type MySQL possédant une table nommée "personnes".



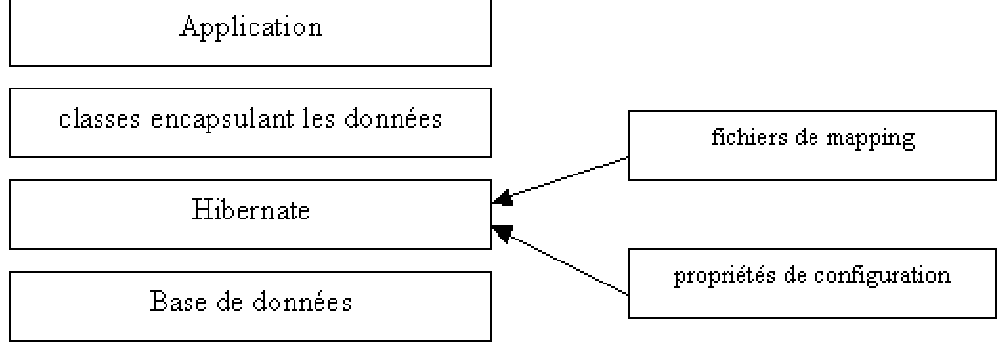
personnes

Field name	Field type
idpersonne *	int(11)
prenompersonne	varchar(50)
nompersonne	varchar(50)
datenaisspersonne	datetime

Hibernate a besoin de plusieurs éléments pour fonctionner :

- une classe de type javabean qui encapsule les données d'une occurrence d'une table
- un fichier de correspondance qui configure la correspondance entre la classe et la table
- des propriétés de configuration notamment des informations concernant la connexion à la base de données

Une fois ces éléments correctement définis, il est possible d'utiliser Hibernate dans le code des traitements à réaliser. L'architecture d'Hibernate est donc la suivante :



Cette section survole uniquement les principales fonctionnalités d'Hibernate qui est un outil vraiment complet : pour de plus amples informations, il est nécessaire de consulter la documentation officielle fournie avec l'outil ou consultable sur le site web.

42.1. La création d'une classe qui va encapsuler les données

Cette classe doit respecter le standard des javabeans notamment encapsuler les propriétés dans ses champs private avec des getters et setters et avoir un constructeur par défaut.

Les types utilisables pour les propriétés sont : les types primitifs, les classes String et Dates, les wrappers, et n'importe quelle classe qui encapsule une autre table ou une partie de la table.

Exemple :

```
import java.util.Date;

public class Personnes {

  private Integer idPersonne;
  private String nomPersonne;
  private String prenomPersonne;
  private Date datenaissPersonne;

  public Personnes(String nomPersonne, String prenomPersonne, Date datenaissPersonne) {
    this.nomPersonne = nomPersonne;
    this.prenomPersonne = prenomPersonne;
    this.datenaissPersonne = datenaissPersonne;
  }

  public Personnes() {
  }

  public Date getDatenaissPersonne() {
    return datenaissPersonne;
  }

  public Integer getIdPersonne() {
    return idPersonne;
  }

  public String getNomPersonne() {
    return nomPersonne;
  }

  public String getPrenomPersonne() {
    return prenomPersonne;
  }

  public void setDatenaissPersonne(Date date) {
    datenaissPersonne = date;
  }

  public void setIdPersonne(Integer integer) {
    idPersonne = integer;
  }

  public void setNomPersonne(String string) {
    nomPersonne = string;
  }

  public void setPrenomPersonne(String string) {
    prenomPersonne = string;
  }

}
```

42.2. La création d'un fichier de correspondance

Pour assurer le mapping, Hibernate a besoin d'un fichier de correspondance (mapping file) au format XML qui va contenir des informations sur la correspondance entre la classe définie et la table de la base de données.

Même si cela est possible, il n'est pas recommandé de définir un fichier de mapping pour plusieurs classes. Le plus simple est de définir un fichier de mapping par classe, nommé du nom de la classe suivi par ".hbm.xml". Ce fichier doit être situé dans le même répertoire que la classe correspondante ou dans la même archive pour les applications packagées.

Différents éléments sont précisés dans ce document XML :

- la classe qui va encapsuler les données
- l'identifiant dans la base de données et son mode de génération
- le mapping entre les propriétés de classe et les champs de la base de données
- les relations
- ...

Le fichier débute par un prologue et une définition de la DTD utilisée par le fichier XML.

Exemple :

```
<?xml version="1.0"?>
  <!DOCTYPE hibernate-mapping
    PUBLIC "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">
```

Le tag racine du document XML est le tag <hibernate-mapping>. Ce tag peut contenir un ou plusieurs tag <class> : il est cependant préférable de n'utiliser qu'un seul tag <class> et de définir autant de fichiers de correspondance que de classes.

Exemple :

Exemple :

```
<?xml version="1.0"?><!DOCTYPE hibernate-mapping
  PUBLIC "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
  "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd"><hibernate-mapping>
  <class name="Personnes" table="personnes">
    <id name="idPersonne" type="int" column="idpersonne">
      <generator class="native"/>
    </id>
    <property name="nomPersonne" type="string" not-null="true" />
    <property name="prenomPersonne" type="string" not-null="true" />
    <property name="datenaissPersonne" type="date">
      <meta attribute="field-description">date de naissance</meta>
    </property>
  </class>
</hibernate-mapping>
```

Le tag <class> permet de préciser des informations sur la classe qui va encapsuler les données.

Ce tag possède plusieurs attributs dont les principaux sont:

Nom	Obligatoire	Rôle
name	oui	nom pleinement qualifié de la classe
table	oui	nom de la table dans la base de données
dynamic-update	non	booléen qui indique de ne mettre à jour que les champs dont la valeur a été modifiée (false par défaut)
dynamic-insert	non	booléen qui indique de ne générer un ordre insert uniquement pour les champs dont la valeur est non nulle (false par défaut)

mutable	non	booléen qui indique si les occurrences peuvent être mises à jour (true par défaut)

Le tag enfant <id> du tag <class> permet de fournir des informations sur l'identifiant d'une occurrence dans la table.

Ce tag possède plusieurs attributs :

Nom	Obligatoire	Rôle
name	non	nom de la propriété dans la classe
type	non	le type Hibernate
column	non	le nom du champ dans la base de données (par défaut le nom de la propriété)
unsaved-value	non	permet de préciser la valeur de l'identifiant pour une instance non encore enregistrée dans la base de données. Les valeurs possibles sont : any, none, null ou une valeur fournie. Null est la valeur par défaut.

Le tag <generator>, fils obligatoire du tag <id>, permet de préciser quel est le mode de génération d'un nouvel identifiant.

Ce tag possède un attribut :

Attribut	Obligatoire	Rôle
class	oui	précise la classe qui va assurer la génération de la valeur d'un nouvel identifiant. Il existe plusieurs classes fournies en standard par Hibernate qui possèdent un nom utilisable comme valeur de cet attribut.

Les classes de génération fournies en standard par Hibernate possèdent chacun un nom :

Nom	Rôle
increment	incrémentation d'une valeur dans la JVM
identity	utilisation d'un identifiant auto-incrémenté pour les bases de données qui le supportent (DB2, MySQL, SQL Server, ...)
sequence	utilisation d'une séquence pour les bases de données qui le supportent (Oracle, DB2, PostgreSQL, ...)
hilo	utilisation d'un algorithme qui utilise une valeur réservée pour une table d'une base de données (par exemple une table qui stocke la valeur du prochain identifiant pour chaque table)
seqhilo	idem mais avec un mécanisme proche d'une séquence
uuid.hex	utilisation d'un algorithme générant un identifiant de type UUID sur 32 caractères prenant en compte entre autre l'adresse IP de la machine et l'heure du système
uuid.string	idem générant un identifiant de type UUID sur 16 caractères
native	utilise la meilleure solution proposée par la base de données
assigned	la valeur est fournie par l'application
foreign	la valeur est fournie par un autre objet avec lequel la classe est associée

Certains modes de génération nécessitent des paramètres : dans ce cas, il faut les définir en utilisant un tag fils <param> pour chaque paramètre.

Le tag <property>, fils du tag <class>, permet de fournir des informations sur une propriété et sa correspondance avec un champ dans la base de données.

Ce tag possède plusieurs attributs dont les principaux sont :

Nom	Obligatoire	Rôle
name	oui	précise le nom de la propriété
type	non	précise le type
column	non	précise le nom du champ dans la base de données (par défaut le nom de la propriété)
update	non	précise si le champ est mis à jour lors d'une opération SQL de type update (par défaut true)
insert	non	précise si le champ est mis à jour lors d'une opération SQL de type insert (par défaut true)

Le type doit être soit un type Hibernate (integer, string, date, timestamp, ...), soit les types primitif Java ou de certaines classes de base (int, java.lang.String, float, java.util.Date, ...), soit une classe qui encapsule des données à rendre persistantes.

Le fichier de correspondance peut aussi contenir une description des relations qui existent avec la table dans la base de données.

42.3. Les propriétés de configuration

Pour exécuter Hibernate, il faut lui fournir un certain nombre de propriétés concernant sa configuration pour qu'il puisse se connecter à la base de données.

Ces propriétés peuvent être fournies sous plusieurs formes :

- un fichier de configuration nommé hibernate.properties et stocké dans un répertoire inclus dans le classpath
- un fichier de configuration au format XML nommé hibernate.cfg.xml
- utiliser la méthode setProperties() de la classe Configuration
- définir des propriétés dans la JVM en utilisant l'option -Dpropriété=valeur

Les principales propriétés pour configurer la connexion JDBC sont :

Nom de la propriété	Rôle
hibernate.connection.driver_class	nom pleinement qualifié de la classe du pilote JDBC
hibernate.connection.url	URL JDBC désignant la base de données
hibernate.connection.username	nom de l'utilisateur pour la connexion
hibernate.connection.password	mot de passe de l'utilisateur
hibernate.connection.pool_size	nombre maximum de connexions dans le pool

Les principales propriétés pour configurer une source de données (DataSource) à utiliser sont :

Nom de la propriété	Rôle
hibernate.connection.datasource	nom du DataSource enregistré dans JNDI
hibernate.jndi.url	URL du fournisseur JNDI
hibernate.jndi.class	classe pleinement qualifiée de type InitialContextFactory permettant l'accès à JNDI
hibernate.connection.username	nom de l'utilisateur de la base de données
hibernate.connection.password	mot de passe de l'utilisateur

Les principales autres propriétés sont :

Nom de la propriété	Rôle
hibernate.dialect	nom de la classe pleinement qualifiée qui assure le dialogue avec la base de données
hibernate.jdbc.use_scrollable_resultset	booléen qui permet le parcours dans les deux sens pour les connexions fournies à Hibernate utilisant des pilotes JDBC 2 supportant cette fonctionnalité
hibernate.show_sql	booléen qui précise si les requêtes SQL générées par Hibernate sont affichées dans la console (particulièrement utile lors du débogage)

Hibernate propose des classes qui héritent de la classe Dialect pour chaque base de données supportée. C'est le nom de la classe correspondant à la base de données utilisée qui doit être obligatoirement fourni à la propriété hibernate.dialect.

Pour définir les propriétés utiles, le plus simple est de définir un fichier de configuration qui en standard doit se nommer hibernate.properties. Ce fichier contient des paires clé=valeur pour chaque propriété définie.

Exemple : paramètres pour utiliser une base de données MySQL

```
hibernate.dialect=net.sf.hibernate.dialect.MySQLDialect
hibernate.connection.driver_class=com.mysql.jdbc.Driver
hibernate.connection.url=jdbc:mysql://localhost/testDB
hibernate.connection.username=root
hibernate.connection.password=
```

Le pilote de la base de données utilisée, mysql-connector-java-3.0.11-stable-bin.jar dans l'exemple, doit être ajouté dans le classpath.

Il est aussi possible de définir les propriétés dans un fichier au format XML nommé en standard hibernate.cfg.xml

Les propriétés sont alors définies par un tag <property>. Le nom de la propriété est fourni grâce à l'attribut « name » et sa valeur est fourni dans le corps du tag.

Exemple :

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 2.0//EN"
  "http://hibernate.sourceforge.net/hibernate-configuration-2.0.dtd">
<hibernate-configuration>
  <session-factory>
    <property name="hibernate.dialect">net.sf.hibernate.dialect.MySQLDialect</property>
    <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="hibernate.connection.url">jdbc:mysql://localhost/testDB</property>
    <property name="hibernate.connection.username">root</property>
    <property name="hibernate.connection.password"></property>
    <property name="dialect">net.sf.hibernate.dialect.MySQLDialect</property>
    <property name="show_sql">true</property>
    <mapping resource="Personnes.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

42.4. L'utilisation d'Hibernate

Pour utiliser Hibernate dans le code, il est nécessaire de réaliser plusieurs opérations :

- création d'une instance de la classe
- création d'une instance de la classe SessionFactory

* création d'une instance de la classe Session qui va permettre d'utiliser les services d'Hibernate

Si les propriétés sont définies dans le fichier hibernate.properties, il faut tout d'abord créer une instance de la classe Configuration. Pour lui associer la ou les classes encapsulant les données, la classe propose deux méthodes :

* addFile() qui attend en paramètre le nom du fichier de mapping
* addClass() qui attend en paramètre un objet de type Class encapsulant la classe. Dans ce cas, la méthode va rechercher un fichier nommé nom_de_la_classe.hbm.xml dans le classpath (ce fichier doit se situer dans le même répertoire que le fichier .class de la classe correspondante)

Une instance de la classe Session est obtenue à partir d'une fabrique de type SessionFactory. Cet objet est obtenu à partir de l'instance du type Configuration en utilisant la méthode buildSessionFactory().

La méthode openSession() de la classe SessionFactory permet d'obtenir une instance de la classe Session.

Par défaut, c'est la méthode openSession() qui va ouvrir une connexion vers la base de données en utilisant les informations fournies par les propriétés de configuration.

Exemple :

```
import net.sf.hibernate.*;
import net.sf.hibernate.cfg.Configuration;
import java.util.Date;

public class TestHibernate1 {

  public static void main(String args[]) throws Exception {
    Configuration config = new Configuration();
    config.addClass(Personnes.class);
    SessionFactory sessionFactory = config.buildSessionFactory();
    Session session = sessionFactory.openSession();
...
```

Il est aussi possible de fournir en paramètre de la méthode openSession() une instance de la classe javax.sql.Connection qui encapsule la connexion à la base de données.

Pour une utilisation du fichier hibernate.cfg.xml, il faut créer une occurrence de la classe Configuration, appeler sa méthode configure() qui va lire le fichier XML et appeler la méthode buildSessionFactory() de l'objet renvoyé par la méthode configure().

Exemple :

```
import net.sf.hibernate.*;
import net.sf.hibernate.cfg.Configuration;
import java.util.*;

public class TestHibernate1 {
  public static void main(String args[]) throws Exception {
    SessionFactory sessionFactory = new Configuration().configure().buildSessionFactory();
...
```

Il est important de clôturer l'objet Session, une fois que celui-ci est devenu inutile, en utilisant la méthode close().

42.5. La persistance d'une nouvelle occurrence

Pour créer une nouvelle occurrence dans la source de données, il suffit de créer une nouvelle instance de la classe encapsulant les données, de valoriser ses propriétés et d'appeler la méthode save() de la session en lui passant en paramètre l'objet encapsulant les données.

La méthode save() n'a aucune action directe sur la base de données. Pour enregistrer les données dans la base, il faut

réaliser un commit sur la connexion ou la transaction ou faire appel à la méthode flush() de la classe Session.

Exemple :

```
import net.sf.hibernate.*;
import net.sf.hibernate.cfg.Configuration;
import java.util.Date;

public class TestHibernate1 {

  public static void main(String args[]) throws Exception {
    Configuration config = new Configuration();
    config.addClass(Personnes.class);
    SessionFactory sessionFactory = config.buildSessionFactory();
    Session session = sessionFactory.openSession();

    Transaction tx = null;
    try {
      tx = session.beginTransaction();
      Personnes personne = new Personnes("nom3", "prenom3", new Date());
      session.save(personne);
      session.flush() ;
      tx.commit();
    } catch (Exception e) {
      if (tx != null) {
        tx.rollback();
      }
      throw e;
    } finally {
      session.close();
    }

    sessionFactory.close();
  }
}
```

Résultat :

```
C:\java\test\testhibernate>ant TestHibernate1
Buildfile: build.xml
init:
[copy] Copying 1 file to C:\java\test\testhibernate\bin
compile:
TestHibernate1:
[java] 12:41:37,402 INFO Environment:462 - Hibernate 2.1.2
[java] 12:41:37,422 INFO Environment:496 - loaded properties from resource
hibernate.properties: {hibernate.connection.username=root, hibernate.connection
.password=, hibernate.cglib.use_reflection_optimizer=true, hibernate.dialect=net
.sf.hibernate.dialect.MySQLDialect, hibernate.connection.url=jdbc:mysql://localh
ost/testDB, hibernate.connection.driver_class=com.mysql.jdbc.Driver}
[java] 12:41:37,432 INFO Environment:519 - using CGLIB reflection optimize
r
[java] 12:41:37,502 INFO Configuration:329 - Mapping resource: Personnes.h
bm.xml
[java] 12:41:38,784 INFO Binder:229 - Mapping class: Personnes -> personne
s
[java] 12:41:38,984 INFO Configuration:595 - processing one-to-many associ
ation mappings
[java] 12:41:38,994 INFO Configuration:604 - processing one-to-one associa
tion property references
[java] 12:41:38,994 INFO Configuration:629 - processing foreign key constr
aints
[java] 12:41:39,074 INFO Dialect:82 - Using dialect: net.sf.hibernate.dial
ect.MySQLDialect
[java] 12:41:39,084 INFO SettingsFactory:62 - Use outer join fetching: tru
e
[java] 12:41:39,104 INFO DriverManagerConnectionProvider:41 - Using Hibern
ate built-in connection pool (not for production use!)
[java] 12:41:39,114 INFO DriverManagerConnectionProvider:42 - Hibernate co
nnection pool size: 20
[java] 12:41:39,144 INFO DriverManagerConnectionProvider:71 - using driver
: com.mysql.jdbc.Driver at URL: jdbc:mysql://localhost/testDB
[java] 12:41:39,154 INFO DriverManagerConnectionProvider:72 - connection p
```

```
roperties: {user=root, password=}
[java] 12:41:39,185 INFO TransactionManagerLookupFactory:33 - No Transacti
onManagerLookup configured (in JTA environment, use of process level read-write
cache is not recommended)
[java] 12:41:39,625 INFO SettingsFactory:102 - Use scrollable result sets:
true
[java] 12:41:39,635 INFO SettingsFactory:105 - Use JDBC3 getGeneratedKeys(
): true
[java] 12:41:39,635 INFO SettingsFactory:108 - Optimize cache for minimal
puts: false
[java] 12:41:39,635 INFO SettingsFactory:117 - Query language substitution
s: {}
[java] 12:41:39,645 INFO SettingsFactory:128 - cache provider: net.sf.ehca
che.hibernate.Provider
[java] 12:41:39,685 INFO Configuration:1080 - instantiating and configurin
g caches
[java] 12:41:39,946 INFO SessionFactoryImpl:119 - building session factory
[java] 12:41:41,237 INFO SessionFactoryObjectFactory:82 - no JNDI name con
figured
[java] 12:41:41,768 INFO SessionFactoryImpl:531 - closing
[java] 12:41:41,768 INFO DriverManagerConnectionProvider:137 - cleaning up
connection pool: jdbc:mysql://localhost/testDB
BUILD SUCCESSFUL
Total time: 7 seconds
C:\java\test\testhibernate>
```

42.6. L'obtention d'une occurrence à partir de son identifiant

La méthode load() de la classe Session permet d'obtenir une instance de la classe des données encapsulant les données de l'occurrence de la base dont l'identifiant est fourni en paramètre.

Il existe deux surcharges de la méthode :

- la première attend en premier paramètre le type de la classe des données et renvoie une nouvelle instance de cette classe
- la seconde attend en paramètre une instance de la classe des données et la met à jour avec les données retrouvées

Exemple :

```
import net.sf.hibernate.*;
import net.sf.hibernate.cfg.Configuration;

public class TestHibernate2 {

  public static void main(String args[]) throws Exception {
    Configuration config = new Configuration();
    config.addClass(Personnes.class);
    SessionFactory sessionFactory = config.buildSessionFactory();
    Session session = sessionFactory.openSession();

    try {
      Personnes personne = (Personnes) session.load(Personnes.class, new Integer(3));
      System.out.println("nom = " + personne.getNomPersonne());
    } finally {
      session.close();
    }

    sessionFactory.close();
  }
}
```

Résultat :

```
C:\java\test\testhibernate>ant TestHibernate2
Buildfile: build.xml
```

```
init:

compile:
    [javac] Compiling 1 source file to C:\java\test\testhibernate\bin

TestHibernate2:
     [java] nom = nom3

BUILD SUCCESSFUL
Total time: 9 seconds
```

42.7. Le langage de requête HQL

Pour offrir un langage d'interrogation commun à toutes les base de données, Hibernate propose son propre langage nommé HQL (Hibernate Query Language)

Le langage HQL est proche de SQL avec une utilisation sous forme d'objets des noms de certaines entités : il n'y a aucune référence aux tables ou aux champs car ceux ci sont référencés respectivement par leur classe et leurs propriétés. C'est Hibernate qui se charge de générer la requête SQL à partir de la requête HQL en tenant compte du contexte (type de base de données utilisée défini dans le fichier de configuration et la configuration du mapping).

La méthode find() de la classe Session permet d'effectuer une recherche d'occurrences grâce à la requête fournie en paramètre.

Exemple : rechercher toutes les occurrences d'une table

```
import net.sf.hibernate.*;
import net.sf.hibernate.cfg.Configuration;
import java.util.*;

public class TestHibernate3 {

  public static void main(String args[]) throws Exception {
    Configuration config = new Configuration();
    config.addClass(Personnes.class);
    SessionFactory sessionFactory = config.buildSessionFactory();
    Session session = sessionFactory.openSession();

    try {
      List personnes = session.find("from Personnes");
      for (int i = 0; i < personnes.size(); i++) {
        Personnes personne = (Personnes) personnes.get(i);
        System.out.println("nom = " + personne.getNomPersonne());
      }
    } finally {
      session.close();
    }
    sessionFactory.close();
  }
}
```

Résultat :

```
C:\java\test\testhibernate>ant TestHibernate3
Buildfile: build.xml

init:

compile:
    [javac] Compiling 1 source file to C:\java\test\testhibernate\bin

TestHibernate3:
     [java] nom = nom1
```

```
     [java] nom = nom2
     [java] nom = nom3

BUILD SUCCESSFUL
Total time: 14 seconds
```

La méthode find() possède deux surcharges pour permettre de fournir un seul ou plusieurs paramètres dans la requête.

La première surcharge permet de fournir un seul paramètre : elle attend en paramètre la requête, la valeur du paramètre et le type du paramètre.

Exemple :

```
import net.sf.hibernate.*;
import net.sf.hibernate.cfg.Configuration;
import java.util.*;

public class TestHibernate4 {

  public static void main(String args[]) throws Exception {
    Configuration config = new Configuration();
    config.addClass(Personnes.class);
    SessionFactory sessionFactory = config.buildSessionFactory();
    Session session = sessionFactory.openSession();

    try {
      List personnes = session.find("from Personnes p where p.nomPersonne=?",
        "nom1", Hibernate.STRING);
      for (int i = 0; i < personnes.size(); i++) {
        Personnes personne = (Personnes) personnes.get(i);
        System.out.println("nom = " + personne.getNomPersonne());
      }
    } finally {
      session.close();
    }

    sessionFactory.close();
  }
}
```

Résultat :

```
C:\java\test\testhibernate>ant TestHibernate4
Buildfile: build.xml

init:

compile:
    [javac] Compiling 1 source file to C:\java\test\testhibernate\bin

TestHibernate4:
     [java] nom = nom1

BUILD SUCCESSFUL
```

Dans la requête du précédent exemple, un alias nommé « p » est défini pour la classe Personnes. Le mode de fonctionnement d'un alias est similaire en HQL et en SQL.

La classe Session propose une méthode iterate() dont le mode de fonctionnement est similaire à la méthode find() mais elle renvoie un itérateur (objet de type Iterator) sur la collection des éléments retrouvés plutôt que la collection elle même.

Exemple :

```
import net.sf.hibernate.*;
import net.sf.hibernate.cfg.Configuration;
```

```
import java.util.*;

public class TestHibernate6 {

  public static void main(String args[]) throws Exception {
    Configuration config = new Configuration();
    config.addClass(Personnes.class);
    SessionFactory sessionFactory = config.buildSessionFactory();
    Session session = sessionFactory.openSession();

    try {
      Iterator personnes = session.iterate("from Personnes ");
      while (personnes.hasNext()) {
        Personnes personne = (Personnes) personnes.next();
        System.out.println("nom = " + personne.getNomPersonne());
      }
    } finally {
      session.close();
    }

    sessionFactory.close();
  }
}
```

Résultat :

```
C:\java\test\testhibernate>ant TestHibernate6
Buildfile: build.xml

init:

compile:
    [javac] Compiling 1 source file to C:\java\test\testhibernate\bin

TestHibernate6:
     [java] nom = nom1
     [java] nom = nom2
     [java] nom = nom3

BUILD SUCCESSFUL
```

Il est aussi possible d'utiliser la clause « order by » dans une requête HQL pour définir l'ordre de tri des occurrences.

Exemple :

List personnes = session.find("from Personnes p order by p.nomPersonne desc");

Il est possible d'utiliser des fonctions telles que count() pour compter le nombre d'occurrences.

Exemple :

```
import net.sf.hibernate.*;
import net.sf.hibernate.cfg.Configuration;
import java.util.*;

public class TestHibernate5 {

  public static void main(String args[]) throws Exception {
    Configuration config = new Configuration();
    config.addClass(Personnes.class);
    SessionFactory sessionFactory = config.buildSessionFactory();
    Session session = sessionFactory.openSession();

    try {
      int compteur = ( (Integer) session.iterate(
        "select count(*) from Personnes").next() ).intValue();
      System.out.println("compteur = " + compteur);
    } finally {
      session.close();
```

```
    }

    sessionFactory.close();
  }
}
```

Résultat :

```
C:\java\test\testhibernate>ant TestHibernate5
Buildfile: build.xml

init:

compile:
    [javac] Compiling 1 source file to C:\java\test\testhibernate\bin

TestHibernate5:
     [java] compteur = 3

BUILD SUCCESSFUL
```

Il est également possible de définir des requêtes utilisant des paramètres nommés grâce à un objet implémentant l'interface Query. Un objet de type Query est obtenu en invoquant la méthode createQuery() de la classe Session avec comme paramètre la requête HQL.

Dans cette requête, les paramètres sont précisés avec un caractère « : » suivi d'un nom unique.

L'interface Query propose de nombreuses méthodes setXXX() pour associer à chaque paramètre une valeur en fonction du type de la valeur (XXX représente le type). Chacune de ces méthodes possède deux surcharges permettant de préciser le paramètre (à partir de son nom ou de son index dans la requête) et sa valeur.

Pour parcourir la collection des occurrences trouvées, l'interface Query propose la méthode list() qui renvoie une collection de type List ou la méthode iterate() qui renvoie un itérateur sur la collection.

Exemple :

```
import net.sf.hibernate.*;
import net.sf.hibernate.cfg.Configuration;
import java.util.*;

public class TestHibernate8 {

  public static void main(String args[]) throws Exception {
    SessionFactory sessionFactory = new Configuration().configure().buildSessionFactory();
    Session session = sessionFactory.openSession();

    try {
      Query query = session.createQuery("from Personnes p where p.nomPersonne = :nom");
      query.setString("nom", "nom2");
      Iterator personnes = query.iterate();

      while (personnes.hasNext()) {
        Personnes personne = (Personnes) personnes.next();
        System.out.println("nom = " + personne.getNomPersonne());
      }
    } finally {
      session.close();
    }

    sessionFactory.close();
  }
}
```

Résultat :

```
C:\java\test\testhibernate>ant TestHibernate8
Buildfile: build.xml
```

```
init:
     [copy] Copying 1 file to C:\java\test\testhibernate\bin

compile:

TestHibernate8:
     [java] nom = nom2

BUILD SUCCESSFUL
Total time: 7 seconds
```

Hibernate propose également d'externaliser une requête dans le fichier de mapping.

42.8. La mise à jour d'une occurrence

Pour mettre à jour une occurrence dans la source de données, il suffit d'appeler la méthode update() de la session en lui passant en paramètre l'objet encapsulant les données.

Le mode de fonctionnement de cette méthode est similaire à celui de la méthode save().

La méthode saveOrUpdate() laisse Hibernate choisir entre l'utilisation de la méthode save() ou update() en fonction de la valeur de l'identifiant dans la classe encapsulant les données.

42.9. La suppression d'une ou plusieurs occurrences

La méthode delete() de la classe Session permet de supprimer une ou plusieurs occurrences en fonction de la version surchargée de la méthode utilisée.

Pour supprimer une occurrence encapsulée dans une classe, il suffit d'invoquer la classe en lui passant en paramètre l'instance de la classe.

Pour supprimer plusieurs occurrences, voire toutes, il faut passer en paramètre de la méthode delete(), une chaîne de caractères contenant la requête HQL pour préciser les éléments concernés par la suppression.

Exemple : suppression de toutes les occurrences de la table

```
session.delete("from Personnes");
```

42.10. Les relations

Un des fondements du modèle de données relationnelles repose sur les relations qui peuvent intervenir entre une table et une ou plusieurs autres tables ou la table elle même.

Les relations utilisables dans le monde relationnel et le monde objet sont cependant différentes.

Les relations du monde objets possèdent quelques caractéristiques :

- Elles sont réalisées via des références entre objets
- Elles peuvent mettre en oeuvre l'héritage et le polymorphisme

Les relations du monde relationnel possèdent quelques caractéristiques :

- Elles sont gérées par des clés étrangères et des jointures entre tables

Les caractéristiques de ces deux modèles sont assez différentes : le but d'un outil de type ORM comme Hibernate est de permettre de manipuler des entités objets et de masquer au développeur le monde relationnel en assurant un mapping entre les deux mondes. Cependant Hibernate ne garantie pas en automatique la gestion inverse des relations qui devra être à la charge du développeur.

Hibernate propose de transcrire ces relations du modèle relationnel dans le modèle objet. Il supporte plusieurs types de relations :

- relation de type 1 - 1 (one-to-one)
- relation de type 1 - n (many-to-one)
- relation de type n - n (many-to-many)

Dans le fichier de mapping, il est nécessaire de définir les relations entre la table concernée et les tables avec lesquelles elle possède des relations.

Les relations peuvent aussi être définies avec des annotations.

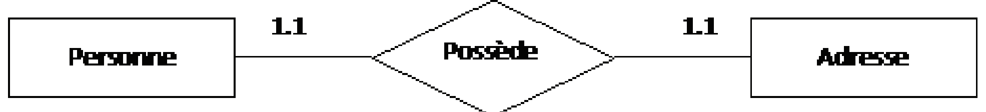
La version d'Hibernate utilisée dans cette section est la 3.5.1.

La base de données utilisée est une base MySQL version 4.1.9.

Chaque exemple possède plusieurs bibliothèques dans son classpath : commons-collections-3.1.jar, dom4j-1.6.1.jar, hibernate-jpa-2.0-api-1.0.0.Final.jar, hibernate3.jar, javassist-3.9.0.GA.jar, jta-1.1.jar, log4j-1.2.15.jar, mysql-connector-java-5.1.12-bin.jar, slf4j-api-1.5.8.jar, slf4j-log4j12-1.5.11.jar.

42.10.1. Les relations un / un

Dans ce type de relation, deux entités sont liées de façon à n'avoir qu'une seule et unique occurrence l'une pour l'autre.



Chaque personne ne peut avoir qu'une seule adresse et une adresse ne peut appartenir qu'à une seule personne.

Cette relation peut se traduire de plusieurs manières dans la base de données :

- Une seule table qui contient les données de la personne et son adresse
- Deux tables, une pour les personnes et une pour les adresses avec une clé primaire partagée
- Deux tables, une pour les personnes et une pour les adresses avec une clé étrangère

Il y a plusieurs façons de traiter ce cas avec une ou deux tables dans la base de données et Hibernate :

- Deux tables et une relation One-to-One d'Hibernate
- Une seule table avec un Component d'Hibernate

42.10.1.1. Le mapping avec un Component

Comme une personne ne peut avoir qu'une seule adresse, il est préférable pour des raisons de performance de stocker les données des deux entités dans une seule et même table. Ceci évite d'avoir à faire une jointure lors de l'accès aux données des deux entités.

La description de la table personne est la suivante :

Résultat :

```
mysql> desc personne;
+------------+--------------+------+-----+---------+----------------+
| Field      | Type         | Null | Key | Default | Extra          |
+------------+--------------+------+-----+---------+----------------+
| Id         | int(11)      |      | PRI | NULL    | auto_increment |
| Nom        | varchar(255) |      |     |         |                |
| Prenom     | varchar(255) |      |     |         |                |
| DateNais   | date         | YES  |     | NULL    |                |
| ligne1_adr | varchar(80)  |      |     |         |                |
| ligne2_adr | varchar(80)  | YES  |     | NULL    |                |
| cp_adr     | varchar(5)   | YES  |     | NULL    |                |
| ville_adr  | varchar(80)  | YES  |     | NULL    |                |
| ligne3_adr | varchar(80)  | YES  |     | NULL    |                |
+------------+--------------+------+-----+---------+----------------+
9 rows in set (0.00 sec)
```

Le script DDL correspondant est le suivant :

Résultat :

```
CREATE TABLE `personne` (
  `Id` int(11) NOT NULL auto_increment,
  `Nom` varchar(255) NOT NULL default '',
  `Prenom` varchar(255) NOT NULL default '',
  `DateNais` date default NULL,
  `ligne1_adr` varchar(80) NOT NULL default '',
  `ligne2_adr` varchar(80) default NULL,
  `cp_adr` varchar(5) default NULL,
  `ville_adr` varchar(80) default NULL,
  `ligne3_adr` varchar(80) default NULL,
  PRIMARY KEY  (`Id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1 AUTO_INCREMENT=0 ;
```

42.10.1.1.1. La configuration dans le fichier de mapping

Les classes qui encapsulent l'entité personne et les données de l'adresse sont de simples POJO.

Exemple :

```
package com.jmdoudoux.test.hibernate;

public class Personne {

  private Long id;
  private String nom;
  private String prenom;
  private String dateNais;
  private Adresse adresse;

  public Personne(String nom, String prenom, String dateNais, Adresse adresse) {
    this.nom = nom;
    this.prenom = prenom;
    this.dateNais = dateNais;
    this.adresse = adresse;
  }

  public Personne() {
  }

  public String getNom() {
    return nom;
  }

  public void setNom(String nom) {
    this.nom = nom;
  }
```

```
  public String getPrenom() {
    return prenom;
  }

  public void setPrenom(String prenom) {
    this.prenom = prenom;
  }

  public String getDateNais() {
    return dateNais;
  }

  public void setDateNais(String dateNais) {
    this.dateNais = dateNais;
  }

  public Long getId() {
    return id;
  }

  // Attention le setter est requis par Hibernate
  public void setId(Long id) {
    this.id = id;
  }

  public Adresse getAdresse() {
    return adresse;
  }

  public void setAdresse(Adresse adresse) {
    this.adresse = adresse;
  }

  @Override
  public String toString() {
    return this.id + " : " + this.nom + " " + this.prenom;
  }
}
```

La classe Adresse ne possède pas de champ de type identifiant.

Exemple :

```
package com.jmdoudoux.test.hibernate;

public class Adresse {

  private String ligne1;
  private String ligne2;
  private String cp;
  private String ville;
  private String ligne3;

  public Adresse(String ligne1, String ligne2, String cp, String ville,
      String ligne3) {
    super();
    this.ligne1 = ligne1;
    this.ligne2 = ligne2;
    this.cp = cp;
    this.ville = ville;
    this.ligne3 = ligne3;
  }

  public Adresse() {
  }

  //
  // getter et setter sur les champs de la classe
  //
```

```
}
```

Les données de l'adresse sont encapsulées dans une classe Adresse : la définition des champs de cette classe est faite dans un élément de type component du fichier de mapping de l'entité Personne (Personne.hbm.xml).

Exemple :

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name="com.jmdoudoux.test.hibernate.Personne"
table="Personne">
    <id name="id"
column="id">
      <generator class="increment" />
    </id>
    <property name="nom" column="Nom" />
    <property name="prenom" column="Prenom" />
    <property name="dateNais" column="DateNais" />
    <component name="adresse" class="com.jmdoudoux.test.hibernate.Adresse">
      <property name="ligne1" column="ligne1_adr" />
      <property name="ligne2" column="ligne2_adr" />
      <property name="cp" column="cp_adr" />
      <property name="ville" column="ville_adr" />
      <property name="ligne3" column="ligne3_adr" />
    </component>
  </class>
</hibernate-mapping>
```

Le fichier de configuration d'Hibernate définit les paramètres de connexion à la base de données et le fichier de mapping de l'entité Personne.

Exemple :

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
          "-//Hibernate/Hibernate Configuration DTD 3.0//EN"

"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <property name="connection.url">jdbc:mysql://localhost/mabdd</property>
    <property name="connection.username">root</property>
    <property name="connection.password"></property>
    <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="transaction.factory_class">
org.hibernate.transaction.JDBCTransactionFactory</property>
    <property name="current_session_context_class">thread</property>
    <property name="hibernate.show_sql">true</property>
    <mapping resource="com/jmdoudoux/test/hibernate/Personne.hbm.xml"></mapping>
  </session-factory>
</hibernate-configuration>
```

L'application de test est basique :

- créer une instance de type Adresse,
- créer une instance de type personne,
- et sauvegarder la personne en base de données

Exemple :

```
package com.jmdoudoux.test.hibernate;
```

```
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;

public class TestHibernate15 {

  public static void main(String args[]) {
    SessionFactory sessionFactory = new Configuration().configure()
                                                        .buildSessionFactory();

    Transaction transaction = null;
    int index = 6;

    Session session = sessionFactory.openSession();

    try {
      transaction = session.beginTransaction();

      Adresse adresse = new Adresse("ligne1_" + index, "ligne2_" + index, "cp_"
          + index, "ville" + index, "ligne3_" + index);
      Personne personne = new Personne("nom" + index,
                                       "prenom_" + index,
                                       null,
                                       adresse);

      session.save(personne);
      transaction.commit();

      System.out.println("La nouvelle personne a ete enregistree");

    } catch (Exception e) {
      transaction.rollback();
      e.printStackTrace();
    } finally {
      session.close();
    }

    sessionFactory.close();
  }
}
```

Lorsque la personne est enregistrée dans la base de données, son adresse l'est aussi dans la table Personne.

Résultat :

```
mysql> select * from personne;
+----+------+----------+----------+-----------+-----------+--------+----------
-+------------+
| Id | Nom  | Prenom   | DateNais | ligne1_adr | ligne2_adr |
cp_adr | ville_adr
 | ligne3_adr |
+----+------+----------+----------+-----------+-----------+--------+----------
-+------------+
|  1 | nom6 | prenom_6 | NULL     | ligne1_6   | ligne2_6
| cp_6   | ville6
 | ligne3_6   |
+----+------+----------+----------+-----------+-----------+--------+----------
-+------------+
1 row in set (0.00 sec)
```

42.10.1.1.2. La configuration avec les annotations

Le POJO qui encapsule une personne a quelques particularités relatives à la relation avec l'adresse :

Il possède un champ privé de type Adresse

Le champ adresse est annoté avec l'annotation @Embedded

Exemple :

```
package com.jmdoudoux.test.hibernate;

import javax.persistence.Column;
import javax.persistence.Embedded;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "personne")
public class Personne {

  @Id
  @GeneratedValue
  @Column(name = "Id")
  private Long id;

  @Column(name = "Nom")
  private String nom;

  @Column(name = "Prenom")
  private String prenom;

  @Column(name = "DateNais")
  private String dateNais;

  @Embedded
  private Adresse adresse;

  public Personne(String nom, String prenom, String dateNais, Adresse adresse) {
    this.nom = nom;
    this.prenom = prenom;
    this.dateNais = dateNais;
    this.adresse = adresse;
  }

  public Personne() {
  }

  public Long getId() {
    return id;
  }

  public Adresse getAdresse() {
    return adresse;
  }

  public void setAdresse(Adresse adresse) {
    this.adresse = adresse;
  }

  //
  // getter et setter sur les autres champs de la classe
  //

  @Override
  public String toString() {
    return this.id + " : " + this.nom + " " + this.prenom;
  }

}
```

L'annotation @Embedded permet de préciser que les données de la classe Adresse seront stockées dans la table Personne comme un component d'Hibernate.

Le POJO qui encapsule une adresse possède plusieurs particularités relatives à la relation avec la personne :

* La classe est annotée avec l'annotation @Embeddable (elle n'est pas annotée comme une entité avec les annotations @Entity et @Table)
* La classe ne possède pas de champ de type identifiant

Exemple :

```
package com.jmdoudoux.test.hibernate;

import javax.persistence.Column;
import javax.persistence.Embeddable;

@Embeddable
public class Adresse {

  @Column(name = "ligne1_adr", nullable = false)
  private String ligne1;

  @Column(name = "ligne2_adr")
  private String ligne2;

  @Column(name = "cp_adr")
  private String cp;

  @Column(name = "ville_adr")
  private String ville;

  @Column(name = "ligne3_adr")
  private String ligne3;

  public Adresse(String ligne1, String ligne2, String cp, String ville,
      String ligne3) {
    super();
    this.ligne1 = ligne1;
    this.ligne2 = ligne2;
    this.cp = cp;
    this.ville = ville;
    this.ligne3 = ligne3;
  }

  public Adresse() {
  }

  //
  // getter et setter sur les champs de la classe
  //

}
```

L'annotation @Embeddable permet de préciser que la classe sera utilisée comme un component. Un tel élément n'a pas d'identifiant puisque celui utilisé sera celui de l'entité englobante.

Le fichier de configuration d'Hibernate définit les paramètres de connexion à la base de données et les deux classes qui encapsulent l'entité Personne et le component Adresse.

Exemple :

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
          "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
          "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <property name="connection.url">jdbc:mysql://localhost/mabdd</property>
    <property name="connection.username">root</property>
    <property name="connection.password"></property>
    <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="transaction.factory_class">
org.hibernate.transaction.JDBCTransactionFactory</property>
    <property name="current_session_context_class">thread</property>
```

```
    <property name="hibernate.show_sql">true</property>
    <mapping class="com.jmdoudoux.test.hibernate.Personne"></mapping>
    <mapping class="com.jmdoudoux.test.hibernate.Adresse"></mapping>
  </session-factory>
</hibernate-configuration>
```

L'application de test est basique :

- créer une instance de type Adresse,
- créer une instance de type Personne,
- et sauvegarder la personne en base de données

Exemple :

```
package com.jmdoudoux.test.hibernate;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.AnnotationConfiguration;

public class TestHibernate16 {

  public static void main(String args[]) {
    SessionFactory sessionFactory = new AnnotationConfiguration().configure()
                                                                 .buildSessionFactory();
    Transaction transaction = null;
    int index = 7;

    Session session = sessionFactory.openSession();

    try {
      transaction = session.beginTransaction();

      Adresse adresse = new Adresse("ligne1_" + index, "ligne2_" + index, "cp_"
          + index, "ville" + index, "ligne3_" + index);
      Personne personne = new Personne("nom" + index,
                                       "prenom_" + index,
                                       null,
                                       adresse);

      session.save(personne);
      transaction.commit();

      System.out.println("La nouvelle personne a ete enregistree");

    } catch (Exception e) {
      transaction.rollback();
      e.printStackTrace();
    } finally {
      session.close();
    }

    sessionFactory.close();
  }
}
```

Lorsque la personne est enregistrée dans la base de données, son adresse l'est aussi dans la table Personne.

Résultat :

```
mysql> select * from personne;
+----+------+----------+----------+-----------+-----------+--------+----------
-+-------------+
| Id | Nom  | Prenom   | DateNais | ligne1_adr | ligne2_adr | cp_adr | ville_adr
 | ligne3_adr |
+----+------+----------+----------+-----------+-----------+--------+----------
-+-------------+
```

```
|  2 | nom7 | prenom_7 | NULL     | ligne1_7   | ligne2_7   | cp_7   | ville7
 | ligne3_7   |
+----+------+----------+----------+------------+------------+--------+----------
-+------------+
1 row in set (0.00 sec)
```

42.10.1.2. Le mapping avec une relation One-to-One avec clé primaire partagée

La relation repose sur deux tables distinctes : une pour les personnes et une pour les adresses.

Chacune des deux tables possède un identifiant qui est sa clé primaire. La particularité est que la valeur des clés primaires est partagée entre les deux tables. L'identifiant de la table adresse n'est pas auto incrémenté et correspond à la valeur de l'identifiant de la table personne.

Hibernate ne sait pas gérer seul ce type de mapping : il sera nécessaire de l'aider en utilisant un mapping bidirectionnel qui permettra à Hibernate de connaitre la valeur de l'identifiant de la personne à utiliser comme valeur de l'identifiant pour l'adresse afin que celui-ci corresponde.

La description de la table personne est la suivante :

Résultat :

```
mysql> desc personne;
+----------+--------------+------+-----+---------+----------------+
| Field    | Type         | Null | Key | Default | Extra          |
+----------+--------------+------+-----+---------+----------------+
| Id       | bigint(20)   |      | PRI | NULL    | auto_increment |
| Nom      | varchar(255) |      |     |         |                |
| Prenom   | varchar(255) |      |     |         |                |
| DateNais | date         | YES  |     | NULL    |                |
+----------+--------------+------+-----+---------+----------------+
4 rows in set (0.00 sec)
```

Le script DDL correspondant est le suivant :

Résultat :

```
CREATE TABLE `personne` (
  `Id` bigint(20) NOT NULL auto_increment,
  `Nom` varchar(255) NOT NULL default '',
  `Prenom` varchar(255) NOT NULL default '',
  `DateNais` date default NULL,
  PRIMARY KEY  (`Id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1 AUTO_INCREMENT=0 ;
```

La description de la table adresse est la suivante :

Résultat :

```
mysql> desc adresse;
+------------+-------------+------+-----+---------+-------+
| Field      | Type        | Null | Key | Default | Extra |
+------------+-------------+------+-----+---------+-------+
| id         | bigint(20)  |      | PRI | 0       |       |
| ligne1_adr | varchar(80) |      |     |         |       |
| ligne2_adr | varchar(80) | YES  |     | NULL    |       |
| cp_adr     | varchar(5)  | YES  |     | NULL    |       |
| ville_adr  | varchar(80) | YES  |     | NULL    |       |
| ligne3_adr | varchar(80) | YES  |     | NULL    |       |
+------------+-------------+------+-----+---------+-------+
6 rows in set (0.11 sec)
```

Le script DDL correspondant est le suivant :

Résultat :

```
CREATE TABLE `adresse` (
  `id` bigint(20) NOT NULL default '0',
  `ligne1_adr` varchar(80) NOT NULL default '',
  `ligne2_adr` varchar(80) default NULL,
  `cp_adr` varchar(5) default NULL,
  `ville_adr` varchar(80) default NULL,
  `ligne3_adr` varchar(80) default NULL,
  PRIMARY KEY  (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

42.10.1.2.1. La configuration dans le fichier de mapping

Les classes qui encapsulent les entités personne et adresse sont de simples POJO.

Exemple :

```
package com.jmdoudoux.test.hibernate;

public class Adresse {

  private Long id;
  private String ligne1;
  private String ligne2;
  private String cp;
  private String ville;
  private String ligne3;

  private Personne personne;

  public Adresse(String ligne1, String ligne2, String cp, String ville,
      String ligne3) {
    super();
    this.ligne1 = ligne1;
    this.ligne2 = ligne2;
    this.cp = cp;
    this.ville = ville;
    this.ligne3 = ligne3;
  }

  public Adresse() {
  }

  public Long getId() {
    return id;
  }

  // setter requis par Hibernate
  public void setId(Long id) {
    this.id = id;
  }

  public Personne getPersonne() {
    return personne;
  }

  public void setPersonne(Personne personne) {
    this.personne = personne;
  }

  //
  // getter et setter sur les autres champs de la classe
  //
```

```
}
```

Exemple :

```
package com.jmdoudoux.test.hibernate;

public class Personne {

  private Long id;
  private String nom;
  private String prenom;
  private String dateNais;
  private Adresse adresse;

  public Personne(String nom, String prenom, String dateNais, Adresse adresse) {
    this.nom = nom;
    this.prenom = prenom;
    this.dateNais = dateNais;
    this.adresse = adresse;
  }

  public Personne() {
  }

  public Long getId() {
    return id;
  }

  // Attention le setter est requis par Hibernate
  public void setId(Long id) {
    this.id = id;
  }

  public Adresse getAdresse() {
    return adresse;
  }

  public void setAdresse(Adresse adresse) {
    this.adresse = adresse;
  }

  //
  // getter et setter sur les autres champs de la classe
  //

  @Override
  public String toString() {
    return this.id + " : " + this.nom + " " + this.prenom;
  }

}
```

Le fichier de mapping de l'entité Personne (Personne.hbm.xml) contient un élément fils <one-to-one> pour définir la relation entre Personne et Adresse.

Exemple :

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name="com.jmdoudoux.test.hibernate.Personne" table="Personne">
    <id name="id" column="id">
      <generator class="increment" />
    </id>
    <property name="nom" column="Nom" />
    <property name="prenom" column="Prenom" />
    <property name="dateNais" column="DateNais" />
```

```
    <one-to-one name="adresse" class="com.jmdoudoux.test.hibernate.Adresse"
      cascade="save-update" />
  </class>
</hibernate-mapping>
```

Le fichier de mapping de l'entité Adresse (Adresse.hbm.xml) possède plusieurs caractéristiques liées au type de la relation utilisée avec l'entité Personne :

- Le champ identifiant id est défini avec un générateur de type foreign avec un paramètre qui précise que la valeur sera celle de l'identifiant du champ personne
- La relation inverse avec Personne est définie avec un tag <one-to-one> avec l'attribut constrained ayant la valeur true

Exemple :

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name="com.jmdoudoux.test.hibernate.Adresse" table="Adresse">

    <id name="id" column="Id">
      <generator class="foreign">
        <param name="property">personne</param>
      </generator>
    </id>
    <property name="ligne1" column="ligne1_adr" />
    <property name="ligne2" column="ligne2_adr" />
    <property name="cp" column="cp_adr" />
    <property name="ville" column="ville_adr" />
    <property name="ligne3" column="ligne3_adr" />
    <one-to-one name="personne" class="com.jmdoudoux.test.hibernate.Personne"
      constrained="true" />
  </class>
</hibernate-mapping>
```

Le fichier de configuration d'Hibernate définit les paramètres de connexion à la base de données et les deux fichiers de mapping des entités.

Exemple :

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
          "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
          "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
<session-factory>
    <property name="connection.url">jdbc:mysql://localhost/mabdd</property>
    <property name="connection.username">root</property>
    <property name="connection.password"></property>
    <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="transaction.factory_class">
org.hibernate.transaction.JDBCTransactionFactory</property>
    <property name="current_session_context_class">thread</property>
    <property name="hibernate.show_sql">true</property>
    <mapping resource="com/jmdoudoux/test/hibernate/Personne.hbm.xml"></mapping>
    <mapping resource="com/jmdoudoux/test/hibernate/Adresse.hbm.xml"></mapping>

</session-factory>
</hibernate-configuration>
```

L'application de test est basique :

* créer une instance de type Adresse,
* créer une instance de type Personne,
* assurer le bon fonctionnement du lien bidirectionnel en fournissant une référence de l'objet Personne à l'instance de l'adresse. Ceci doit être fait manuellement car Hibernate ne prend pas en charge automatiquement les liens bidirectionnels
* et sauvegarder la personne en base de données

Exemple :

```
package com.jmdoudoux.test.hibernate;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;

public class TestHibernate11 {

  public static void main(String args[]) {
    SessionFactory sessionFactory = new Configuration().configure()
                                                     .buildSessionFactory();
    Transaction transaction = null;
    int index = 3;

    Session session = sessionFactory.openSession();

    try {
      transaction = session.beginTransaction();

      Adresse adresse = new Adresse("ligne1_" + index, "ligne2_" + index, "cp_"
          + index, "ville" + index, "ligne3_" + index);
      Personne personne = new Personne("nom" + index,
                                       "prenom_" + index,
                                       null,
                                       adresse);
      adresse.setPersonne(personne);

      session.save(personne);
      transaction.commit();

      System.out.println("La nouvelle personne a ete enregistree");

    } catch (Exception e) {
      transaction.rollback();
      e.printStackTrace();
    } finally {
      session.close();
    }

    sessionFactory.close();
  }
}
```

Lorsque la personne est enregistrée dans la base de données, son adresse l'est aussi avec comme identifiant la valeur de l'identifiant de la personne.

Résultat :

```
mysql> select * from personne;
+----+------+----------+----------+
| Id | Nom  | Prenom   | DateNais |
+----+------+----------+----------+
|  1 | nom3 | prenom_3 | NULL     |
+----+------+----------+----------+
1 row in set (0.00 sec)

mysql> select * from adresse;
+----+------------+------------+--------+-----------+------------+
| id | ligne1_adr | ligne2_adr | cp_adr | ville_adr | ligne3_adr |
+----+------------+------------+--------+-----------+------------+
```

```
|  1 | ligne1_3    | ligne2_3    | cp_3    | ville3     | ligne3_3    |
+----+------------+------------+--------+-----------+------------+
1 row in set (0.00 sec)
```

42.10.1.2.2. La configuration avec les annotations

Le POJO qui encapsule une personne a quelques particularités relatives à la relation avec l'adresse :

- Il possède un champ privé de type Adresse
- Le champ adresse est annoté avec les annotations @OneToOne et @PrimaryKeyJoin

Exemple :

```
package com.jmdoudoux.test.hibernate;

import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.OneToOne;
import javax.persistence.PrimaryKeyJoinColumn;
import javax.persistence.Table;

@Entity
@Table(name = "personne")
public class Personne {

  @Id
  @GeneratedValue
  @Column(name = "Id")
  private Long id;

  @Column(name = "Nom")
  private String nom;

  @Column(name = "Prenom")
  private String prenom;

  @Column(name = "DateNais")
  private String dateNais;

  @OneToOne(cascade = CascadeType.ALL)
  @PrimaryKeyJoinColumn
  private Adresse adresse;

  public Personne(String nom, String prenom, String dateNais, Adresse adresse) {
    this.nom = nom;
    this.prenom = prenom;
    this.dateNais = dateNais;
    this.adresse = adresse;
  }

  public Personne() {
  }

  public Long getId() {
    return id;
  }

  public Adresse getAdresse() {
    return adresse;
  }

  public void setAdresse(Adresse adresse) {
    this.adresse = adresse;
  }
```

```
  //
  // getter et setter sur les autres champs de la classe
  //

  @Override
  public String toString() {
    return this.id + " : " + this.nom + " " + this.prenom;
  }

}
```

Si le champ adresse n'est pas annoté avec l'annotation @PrimaryKeyJoin, alors une exception de type org.hibernate.id.IdentifierGenerationException avec le message « null id generated for:class com.jmdoudoux.test.hibernate.Adresse » est levée à l'exécution.

Le POJO qui encapsule une adresse possède plusieurs particularités relatives à la relation avec la personne :

- Le champ identifiant de l'entité est annoté avec @GeneratedValue et @GenericGenerator pour indiquer à Hibernate que la valeur du champ id doit être obtenue à partie de la valeur du champ id de la propriété personne
- Un champ de type Personne permet une relation bidirectionnelle annotée avec @OneToOne
- Un setter sur le champ personne permettra d'assurer la cohésion de la relation par le développeur

Exemple :

```
package com.jmdoudoux.test.hibernate;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.OneToOne;
import javax.persistence.Table;

import org.hibernate.annotations.Parameter;

@Entity
@Table(name = "adresse")
public class Adresse {

  @Id
  @GeneratedValue(generator = "adresseGenerator")
  @org.hibernate.annotations.GenericGenerator(name = "adresseGenerator",
    strategy = "foreign", parameters = @Parameter(name = "property", value = "personne"))
  @Column(name = "id")
  private Long id;

  @Column(name = "ligne1_adr", nullable = false)
  private String ligne1;

  @Column(name = "ligne2_adr")
  private String ligne2;

  @Column(name = "cp_adr")
  private String cp;

  @Column(name = "ville_adr")
  private String ville;

  @Column(name = "ligne3_adr")
  private String ligne3;

  @OneToOne(mappedBy = "adresse")
  private Personne personne;

  public Adresse(String ligne1, String ligne2, String cp, String ville,
      String ligne3) {
    super();
    this.ligne1 = ligne1;
    this.ligne2 = ligne2;
    this.cp = cp;
```

```
    this.ville = ville;
    this.ligne3 = ligne3;
  }

  public Adresse() {
  }

  public Long getId() {
    return id;
  }

  public Personne getPersonne() {
    return personne;
  }

  public void setPersonne(Personne personne) {
    this.personne = personne;
  }

  //
  // getter et setter sur les autres champs de la classe
  //

}
```

Le fichier de configuration d'Hibernate définit les paramètres de connexion à la base de données et les deux classes qui encapsulent des entités.

Exemple :

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
          "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
          "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
<session-factory>
    <property name="connection.url">jdbc:mysql://localhost/mabdd</property>
    <property name="connection.username">root</property>
    <property name="connection.password"></property>
    <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="transaction.factory_class">
org.hibernate.transaction.JDBCTransactionFactory</property>
    <property name="current_session_context_class">thread</property>
    <property name="hibernate.show_sql">true</property>
    <mapping class="com.jmdoudoux.test.hibernate.Personne"></mapping>
    <mapping class="com.jmdoudoux.test.hibernate.Adresse"></mapping>

</session-factory>
</hibernate-configuration>
```

L'application de test est basique :

- créer une instance de type Adresse,
- créer une instance de type Personne,
- assurer le bon fonctionnement du lien bidirectionnel en fournissant une référence de l'objet Personne à l'instance de l'adresse. Ceci doit être fait manuellement car Hibernate ne prend pas en charge automatiquement les liens bidirectionnels
- et sauvegarder la personne en base de données

Exemple :

```
package com.jmdoudoux.test.hibernate;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
```

```
import org.hibernate.cfg.AnnotationConfiguration;

public class TestHibernate10 {

  public static void main(String args[]) {
    SessionFactory sessionFactory = new AnnotationConfiguration().configure()
                                                                 .buildSessionFactory();
    Transaction transaction = null;
    int index = 2;

    Session session = sessionFactory.openSession();

    try {
      transaction = session.beginTransaction();

      Adresse adresse = new Adresse("ligne1_" + index, "ligne2_" + index, "cp_"
          + index, "ville" + index, "ligne3_" + index);
      Personne personne = new Personne("nom" + index,
                                       "prenom_" + index,
                                       null,
                                       adresse);
      adresse.setPersonne(personne);

      session.save(personne);
      transaction.commit();

      System.out.println("La nouvelle personne a ete enregistree");

    } catch (Exception e) {
      transaction.rollback();
      e.printStackTrace();
    } finally {
      session.close();
    }

    sessionFactory.close();
  }
}
```

Lorsque la personne est enregistrée dans la base de données, son adresse l'est aussi avec comme identifiant la valeur de l'identifiant de la personne.

Résultat :

```
mysql> select * from adresse;
+----+------------+------------+--------+-----------+------------+
| id | ligne1_adr | ligne2_adr | cp_adr | ville_adr | ligne3_adr |
+----+------------+------------+--------+-----------+------------+
|  3 | ligne1_1   | ligne2_1   | cp_1   | ville1    | ligne3_1   |
+----+------------+------------+--------+-----------+------------+
1 row in set (0.00 sec)

mysql> select * from personne;
+----+------+----------+----------+
| Id | Nom  | Prenom   | DateNais |
+----+------+----------+----------+
|  3 | nom1 | prenom_1 | NULL     |
+----+------+----------+----------+
1 row in set (0.00 sec)
```

Si la référence de l'instance de type Personne n'est pas fournie à l'instance de type Adresse alors une exception de type org.hibernate.id.IdentifierGenerationException avec le message « attempted to assign id from null one-to-one property [com.jmdoudoux.test.hibernate.Adresse.personne] » est levée à l'exécution.

Si le générateur d'identifiant n'est pas correctement configuré pour l'entité Adresse, alors une exception de type org.hibernate.id.IdentifierGenerationException avec le message « ids for this class must be manually assigned before calling save(): com.jmdoudoux.test.hibernate.Adresse » lors de l'exécution.

42.10.1.3. Le mapping avec une relation One-to-One avec clé étrangère

La relation repose sur deux tables distinctes : une pour les personnes et une pour les adresses

Chacune des deux tables possède son propre identifiant et la relation entre les deux tables est assurée par une clé étrangère de la table personne vers la table adresse.

Hibernate sait gérer seul ce type de mapping si il est unidirectionnel.

La description de la table personne est la suivante :

Résultat :

```
mysql> desc personne;
+------------+--------------+------+-----+---------+----------------+
| Field      | Type         | Null | Key | Default | Extra          |
+------------+--------------+------+-----+---------+----------------+
| Id         | int(11)      |      | PRI | NULL    | auto_increment |
| Nom        | varchar(255) |      |     |         |                |
| Prenom     | varchar(255) |      |     |         |                |
| DateNais   | date         | YES  |     | NULL    |                |
| adresse_id | int(11)      |      |     | 0       |                |
+------------+--------------+------+-----+---------+----------------+
5 rows in set (0.00 sec)
```

Le script DDL correspondant est le suivant :

Résultat :

```
CREATE TABLE `personne` (
  `Id` int(11) NOT NULL auto_increment,
  `Nom` varchar(255) NOT NULL default '',
  `Prenom` varchar(255) NOT NULL default '',
  `DateNais` date default NULL,
  `adresse_id` int(11) NOT NULL default '0',
  PRIMARY KEY  (`Id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1 AUTO_INCREMENT=0 ;
```

La description de la table adresse est la suivante :

Résultat :

```
mysql> desc adresse;
+------------+-------------+------+-----+---------+----------------+
| Field      | Type        | Null | Key | Default | Extra          |
+------------+-------------+------+-----+---------+----------------+
| id         | bigint(20)  |      | PRI | NULL    | auto_increment |
| ligne1_adr | varchar(80) |      |     |         |                |
| ligne2_adr | varchar(80) | YES  |     | NULL    |                |
| cp_adr     | varchar(5)  | YES  |     | NULL    |                |
| ville_adr  | varchar(80) | YES  |     | NULL    |                |
| ligne3_adr | varchar(80) | YES  |     | NULL    |                |
+------------+-------------+------+-----+---------+----------------+
6 rows in set (0.00 sec)
```

Le script DDL correspondant est le suivant :

Résultat :

```
CREATE TABLE `adresse` (
  `id` bigint(20) NOT NULL auto_increment,
  `ligne1_adr` varchar(80) NOT NULL default '',
```

```
  `ligne2_adr` varchar(80) default NULL,
  `cp_adr` varchar(5) default NULL,
  `ville_adr` varchar(80) default NULL,
  `ligne3_adr` varchar(80) default NULL,
  PRIMARY KEY  (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1 AUTO_INCREMENT=0 ;
```

42.10.1.3.1. La configuration dans le fichier de mapping

Les classes qui encapsulent les entités personne et adresse sont de simples POJO.

Exemple :

```
package com.jmdoudoux.test.hibernate;

public class Personne {

  private Long id;
  private String nom;
  private String prenom;
  private String dateNais;
  private Adresse adresse;

  public Personne(String nom, String prenom, String dateNais, Adresse adresse) {
    this.nom = nom;
    this.prenom = prenom;
    this.dateNais = dateNais;
    this.adresse = adresse;
  }

  public Personne() {
  }

  public Long getId() {
    return id;
  }

  // Attention le setter est requis par Hibernate
  public void setId(Long id) {
    this.id = id;
  }

  public Adresse getAdresse() {
    return adresse;
  }

  public void setAdresse(Adresse adresse) {
    this.adresse = adresse;
  }

  //
  // getter et setter sur les autres champs de la classe
  //

  @Override
  public String toString() {
    return this.id + " : " + this.nom + " " + this.prenom;
  }

}
```

Exemple :

```
package com.jmdoudoux.test.hibernate;
public class Adresse {
  private Long id;
  private String ligne1;
  private String ligne2;
```

```
  private String cp;
  private String ville;
  private String ligne3;
  public Adresse(String ligne1,
String ligne2, String cp, String ville,
      String ligne3) {
    super();
    this.ligne1 = ligne1;
    this.ligne2 = ligne2;
    this.cp = cp;
    this.ville = ville;
    this.ligne3 = ligne3;
  }
  public Adresse() {
  }
  public Long getId() {
    return id;
  }
  // setter requis par Hibernate
  public void setId(Long id) {
    this.id = id;
  }

  //
  // getter et setter sur les autres champs de la classe
  //

}
```

Le fichier de mapping de l'entité Personne (Personne.hbm.xml) contient un élément fils <many-to-one> pour définir la relation entre Personne et Adresse : l'unicité de la relation est cependant garantie par la valeur true de l'attribut unique. Il faut aussi utiliser une propriété column pour préciser la colonne qui va contenir la clé étrangère vers la table adresse.

Exemple :

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name="com.jmdoudoux.test.hibernate.Personne" table="Personne">
    <id name="id" column="Id">
       <generator class="increment" />
    </id>
    <property name="nom" column="Nom" />
    <property name="prenom" column="Prenom" />
    <property name="dateNais" column="DateNais" />
    <many-to-one name="adresse" class="com.jmdoudoux.test.hibernate.Adresse"
        column="adresse_id" cascade="all" unique="true" />
  </class>
</hibernate-mapping>
```

Le fichier de mapping de l'entité Adresse (Adresse.hbm.xml) ne contient aucune particularité.

Exemple :

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name="com.jmdoudoux.test.hibernate.Adresse" table="Adresse">
    <id name="id">
      <generator class="increment" />
    </id>
    <property name="ligne1" column="ligne1_adr" />
    <property name="ligne2" column="ligne2_adr" />
    <property name="cp" column="cp_adr" />
```

```
    <property name="ville" column="ville_adr" />
    <property name="ligne3" column="ligne3_adr" />
  </class>
</hibernate-mapping>
```

Le fichier de configuration d'Hibernate définit les paramètres de connexion à la base de données et les deux fichiers de mapping des entités.

Exemple :

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
          "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
          "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
<session-factory>
    <property name="connection.url">jdbc:mysql://localhost/mabdd</property>
    <property name="connection.username">root</property>
    <property name="connection.password"></property>
    <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="transaction.factory_class">
org.hibernate.transaction.JDBCTransactionFactory</property>
    <property name="current_session_context_class">thread</property>
    <property name="hibernate.show_sql">true</property>
    <mapping resource="com/jmdoudoux/test/hibernate/Personne.hbm.xml"></mapping>
    <mapping resource="com/jmdoudoux/test/hibernate/Adresse.hbm.xml"></mapping>

</session-factory>
</hibernate-configuration>
```

L'application de test est basique :

- créer une instance de type Adresse,
- créer une instance de type Personne,
- et sauvegarder la personne en base de données

Exemple :

```
package com.jmdoudoux.test.hibernate;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;

public class TestHibernate12 {

  public static void main(String args[]) {
    SessionFactory sessionFactory = new Configuration().configure()
                                                       .buildSessionFactory();
    Transaction transaction = null;
    int index = 4;

    Session session = sessionFactory.openSession();

    try {
      transaction = session.beginTransaction();

      Adresse adresse = new Adresse("ligne1_" + index, "ligne2_" + index, "cp_"
          + index, "ville" + index, "ligne3_" + index);
      Personne personne = new Personne("nom_" + index,
                                       "prenom_" + index,
                                       null,
                                       adresse);

      session.save(personne);
      transaction.commit();
```

```
      System.out.println("La nouvelle personne a ete enregistree");

    } catch (Exception e) {
      transaction.rollback();
      e.printStackTrace();
    } finally {
      session.close();
    }

    sessionFactory.close();
  }
}
```

Lorsque la personne est enregistrée dans la base de données, son adresse l'est aussi. La nouvelle occurrence de la table Personne et de la table Adresse possède chacun leur propre identifiant et celui de l'adresse est reporté dans le champ adresse_id de la table Personne.

Résultat :

```
mysql> select * from personne;
+----+-------+----------+----------+------------+
| Id | Nom   | Prenom   | DateNais | adresse_id |
+----+-------+----------+----------+------------+
|  1 | nom_4 | prenom_4 | NULL     |          8 |
+----+-------+----------+----------+------------+
1 row in set (0.82 sec)

mysql> select * from adresse;
+----+------------+------------+--------+-----------+------------+
| id | ligne1_adr | ligne2_adr | cp_adr | ville_adr | ligne3_adr |
+----+------------+------------+--------+-----------+------------+
|  8 | ligne1_4   | ligne2_4   | cp_4   | ville4    | ligne3_4   |
+----+------------+------------+--------+-----------+------------+
1 row in set (0.00 sec)
```

42.10.1.3.2. La configuration avec les annotations

Le POJO qui encapsule une personne a quelques particularités relatives à la relation avec l'adresse :

- Il possède un champ privé de type Adresse
- Le champ adresse est annoté avec les annotations @OneToOne et @JoinColumn

Exemple :

```
package com.jmdoudoux.test.hibernate;

import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.OneToOne;
import javax.persistence.Table;

@Entity
@Table(name = "personne")
public class Personne {

  @Id
  @GeneratedValue
  @Column(name = "Id")
  private Long id;
```

```
  @Column(name = "Nom")
  private String nom;

  @Column(name = "Prenom")
  private String prenom;

  @Column(name = "DateNais")
  private String dateNais;

  @OneToOne(cascade = CascadeType.ALL)
  @JoinColumn(name = "adresse_id")
  private Adresse adresse;

  public Personne(String nom, String prenom, String dateNais, Adresse adresse) {
    this.nom = nom;
    this.prenom = prenom;
    this.dateNais = dateNais;
    this.adresse = adresse;
  }

  public Personne() {
  }

  public Long getId() {
    return id;
  }

  public Adresse getAdresse() {
    return adresse;
  }

  public void setAdresse(Adresse adresse) {
    this.adresse = adresse;
  }

  //
  // getter et setter sur les autres champs de la classe
  //

  @Override
  public String toString() {
    return this.id + " : " + this.nom + " " + this.prenom;
  }
}
```

Le POJO qui encapsule une adresse ne possède aucune particularité relative à la relation avec la personne. Il est cependant possible d'ajouter au besoin une relation inverse d'adresse vers personne en ajoutant un champ Personne annoté avec l'annotation @one-to-one possèdant un attribut mappedBy qui possède comme valeur le nom du champ de l'adresse dans l'entité Personne.

Dans ce cas, la gestion de l'alimentation du champ personne est à la charge du développeur en utilisant le setter sur le champ personne.

L'identifiant de l'entité est annoté avec @GeneratedValue

Exemple :

```
package com.jmdoudoux.test.hibernate;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "adresse")
public class Adresse {

  @Id
```

```
  @GeneratedValue
  @Column(name = "id")
  private Long id;

  @Column(name = "ligne1_adr", nullable = false)
  private String ligne1;

  @Column(name = "ligne2_adr")
  private String ligne2;

  @Column(name = "cp_adr")
  private String cp;

  @Column(name = "ville_adr")
  private String ville;

  @Column(name = "ligne3_adr")
  private String ligne3;

  public Adresse(String ligne1, String ligne2, String cp, String ville,
      String ligne3) {
    super();
    this.ligne1 = ligne1;
    this.ligne2 = ligne2;
    this.cp = cp;
    this.ville = ville;
    this.ligne3 = ligne3;
  }

  public Adresse() {
  }

  public Long getId() {
    return id;
  }

  //
  // getter et setter sur les autres champs de la classe
  //

}
```

Le fichier de configuration d'Hibernate définit les paramètres de connexion à la base de données et les deux classes qui encapsulent des entités.

Exemple :

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
          "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
          "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
<session-factory>
    <property name="connection.url">jdbc:mysql://localhost/mabdd</property>
    <property name="connection.username">root</property>
    <property name="connection.password"></property>
    <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="transaction.factory_class">
org.hibernate.transaction.JDBCTransactionFactory</property>
    <property name="current_session_context_class">thread</property>
    <property name="hibernate.show_sql">true</property>
    <mapping class="com.jmdoudoux.test.hibernate.Personne"></mapping>
    <mapping class="com.jmdoudoux.test.hibernate.Adresse"></mapping>

</session-factory>
</hibernate-configuration>
```

L'application de test est basique :

* créer une instance de type Adresse,
* créer une instance de type Personne,
* et sauvegarder la personne en base de données

Exemple :

```
package com.jmdoudoux.test.hibernate;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.AnnotationConfiguration;

public class TestHibernate14 {

  public static void main(String args[]) {
    SessionFactory sessionFactory = new AnnotationConfiguration().configure()
                                                              .buildSessionFactory();
    Transaction transaction = null;
    int index = 5;

    Session session = sessionFactory.openSession();

    try {
      transaction = session.beginTransaction();

      Adresse adresse = new Adresse("ligne1_" + index, "ligne2_" + index, "cp_"
          + index, "ville" + index, "ligne3_" + index);
      Personne personne = new Personne("nom" + index,
                                       "prenom_" + index,
                                       null,
                                       adresse);
      session.save(personne);
      transaction.commit();

      System.out.println("La nouvelle personne a ete enregistree");

    } catch (Exception e) {
      transaction.rollback();
      e.printStackTrace();
    } finally {
      session.close();
    }

    sessionFactory.close();
  }
}
```

Lorsque la personne est enregistrée dans la base de données, son adresse l'est aussi. La nouvelle occurrence de la table Personne et de la table Adresse possède chacun leur propre identifiant et celui de l'adresse est reporté dans le champ adresse_id de la table Personne.

Résultat :

```
mysql> select * from adresse;
+----+------------+------------+--------+-----------+------------+
| id | ligne1_adr | ligne2_adr | cp_adr | ville_adr | ligne3_adr |
+----+------------+------------+--------+-----------+------------+
| 10 | ligne1_5   | ligne2_5   | cp_5   | ville5    | ligne3_5   |
+----+------------+------------+--------+-----------+------------+
1 row in set (0.00 sec)

mysql> select * from personne;
+----+-------+----------+----------+------------+
| Id | Nom   | Prenom   | DateNais | adresse_id |
+----+-------+----------+----------+------------+
|  8 | nom5  | prenom_5 | NULL     |         10 |
+----+-------+----------+----------+------------+
1 row in set (0.00 sec)
```

La suite de cette section sera développée dans une version future de ce document

42.11. Les outils de génération de code

Hibernate fournit séparément un certain nombre d'outils. Ces outils sont livrés séparément dans un fichier nommé hibernate-extensions-2.1.zip.

Il faut télécharger et décompresser le contenu de cette archive par exemple dans le répertoire ou Hibernate a été décompressé.

L'archive contient deux répertoires :

- hibern8ide
- tools

Le répertoire tools propose trois outils :

- class2hbm :
- ddl2hbm :
- hbm2java :

Pour utiliser ces outils, il y a deux solutions possibles :

- utiliser les fichiers de commande .bat fourni dans le répertoire /tools/bin
- utiliser ant pour lancer ces outils

Pour utiliser les fichiers de commandes .bat, il est nécessaire au préalable de configurer les paramètres dans le fichier setenv.bat. Il faut notamment correctement renseigner les valeurs associées aux variables JDBC_DRIVER qui précise le pilote de la base de données et HIBERNATE_HOME qui précise le répertoire ou est installé Hibernate.

Pour utiliser les outils dans un script ant, il faut créer ou modifier un fichier build en ajoutant une tâche pour l'outil à utiliser.

Il faut copier le fichier hibernate2.jar et les fichiers contenus dans le répertoire /lib d'Hibernate dans le répertoire lib du projet. Il faut aussi copier dans ce répertoire les fichiers contenus dans le répertoire /tools/lib et le fichier /tools/hibernate-tools.jar.

Pour éviter les messages d'avertissement sur la configuration manquante de log4j, le plus simple est de copier le fichier /src/log4j.properties d'Hibernate dans le répertoire bin du projet.

La suite de ce chapitre sera développée dans une version future de ce document

43. JPA (Java Persistence API)

L'utilisation pour la persistance d'un mapping O/R permet de proposer un niveau d'abstraction plus élevé que la simple utilisation de JDBC : ce mapping permet d'assurer la transformation d'objets vers la base de données et vice et versa que cela soit pour des lectures ou des mises à jour (création, modification ou suppression).

Développée dans le cadre de la version 3.0 des EJB, cette API ne se limite pas aux EJB puisqu'elle aussi être mise en oeuvre dans des applications Java SE.

L'utilisation de l'API ne requiert aucune ligne de code mettant en oeuvre l'API JDBC.

L'API propose un langage d'interrogation similaire à SQL mais utilisant des objets plutôt que des entités relationnelles de la base de données.

L'API Java Persistence repose sur des entités qui sont de simples POJOs annotés et sur un gestionnaire de ces entités (EntityManager) qui propose des fonctionnalités pour les manipuler (ajout, modification suppression, recherche). Ce gestionnaire est responsable de la gestion de l'état des entités et de leur persistance dans la base de données.

43.1. L'installation de l'implémentation de référence

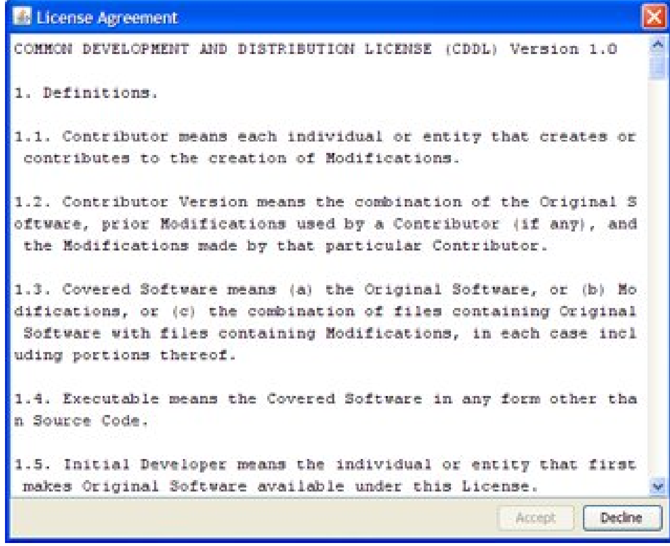
L'implémentation de référence est incluse dans le projet GlassFish. Elle peut être téléchargée unitairement à l'url : https://glassfish.dev.java.net/downloads/persistence/JavaPersistence.html

Cette implémentation de référence repose sur l'outil TopLink d'Oracle dans sa version essential.

Il suffit d'exécuter la commande java -jar avec en paramètre le fichier jar téléchargé.

Exemple :

```
C:\>java -jar glassfish-persistence-installer-v2-b52.jar
glassfish-persistence
glassfish-persistence\README
glassfish-persistence\3RD-PARTY-LICENSE.txt
glassfish-persistence\LICENSE.txt
glassfish-persistence\toplink-essentials-agent.jar
glassfish-persistence\toplink-essentials.jar
installation complete
```



Lisez la licence et si vous l'acceptez, cliquez sur le bouton « Accept ».

Un répertoire glassfish-persistence est créé contenant les bibliothèques de l'implémentation de référence de JPA.

43.2. Les entités

Les entités dans les spécifications de l'API Java Persistence permettent d'encapsuler les données d'une occurrence d'une ou plusieurs tables. Ce sont de simples POJO (Plain Old Java Object). Un POJO est une classe Java qui n'implémente aucune interface particulière ni n'hérite d'aucune classe mère spécifique.

Un objet Java de type POJO mappé vers une table de la base de données grâce à des méta data via l'API Java Persistence est nommé bean entité (Entity bean).

Un bean entité doit obligatoirement avoir un constructeur sans argument et la classe du bean doit obligatoirement être marquée avec l'annotation @javax.persistence.Entity. Cette annotation possède un attribut optionnel nommé name qui permet de préciser le nom de l'entité dans les requêtes. Par défaut, ce nom est celui de la classe de l'entité.

En tant que POJO, le bean entity n'a pas à implémenter d'interface particulière mais il doit en plus de respecter les règles de tous Java beans :

- Être déclaré avec l'annotation @javax.persistence.Entity
- Posséder au moins une propriété déclarer comme clé primaire avec l'annotation @Id

Le bean entity est composé de propriétés qui seront mappés sur les champs de la table de la base de données sous jacente. Chaque propriété encapsule les données d'un champ d'une table. Ces propriétés sont utilisables au travers de simple accesseurs (getter/setter).

Une propriété particulière est la clé primaire qui sert d'identifiant unique dans la base de données mais aussi dans le POJO. Elle peut être de type primitif ou de type objet. La déclaration de cette clé primaire est obligatoire.

43.2.1. Le mapping entre le bean entité et la table

La description du mapping entre le bean entité et la table peut être fait de deux façons :

- Utiliser des annotations
- Utiliser un fichier XML de mapping

L'API propose plusieurs annotations pour supporter un mapping O/R assez complet.

Annotation	Rôle
@javax.persistence.Table	Préciser le nom de la table concernée par le mapping
@javax.persistence.Column	Associé à un getter, il permet d'associer un champ de la table à la propriété
@javax.persistence.Id	Associé à un getter, il permet d'associer un champ de la table à la propriété en tant que clé primaire
@javax.persistence.GeneratedValue	Demander la génération automatique de la clé primaire au besoin
@javax.persistence.Basic	Représenter la forme de mapping la plus simple. Cette annotation est utilisée par défaut
@javax.persistence.Transient	Demander de ne pas tenir compte du champ lors du mapping

L'annotation @javax.persistence.Table permet de lier l'entité à une table de la base de données. Par défaut, l'entité est liée à la table de la base de données correspondant au nom de la classe de l'entité. Si ce nom est différent alors l'utilisation de l'annotation @Table est obligatoire. C'est notamment le cas si des conventions de nommage des entités de la base de données sont mises en place.

L'annotation @Table possède plusieurs attributs :

Attributs	Rôle
name	Nom de la table
catalog	Catalogue de la table
schema	Schéma de la table
uniqueConstraints	Contraintes d'unicité sur une ou plusieurs colonnes

L'annotation @javax.persistence.Column permet d'associer un membre de l'entité à une colonne de la table. Par défaut, les champs de l'entité sont liés aux champs de la table dont les noms correspondent. Si ces noms sont différents alors l'utilisation de l'annotation @Column est obligatoire. C'est notamment le cas si des conventions de nommage des entités de la base de données sont mises en place.

L'annotation @Column possède plusieurs attributs :

Attributs	Rôle
name	Nom de la colonne
table	Nom de la table dans le cas d'un mapping multi-table
unique	Indique si la colonne est unique
nullable	Indique si la colonne est nullable
insertable	Indique si la colonne doit être prise en compte dans les requêtes de type insert
updatable	Indique si la colonne doit être prise en compte dans les requêtes de type update
columnDefinition	Précise le DDL de définition de la colonne
length	Indique la taille d'une colonne de type chaîne de caractères
precision	Indique la taille d'une colonne de type numérique
scale	Indique la précision d'une colonne de type numérique

Hormis les attributs name et table, tous les autres attributs ne sont utilisés que par un éventuel outil du fournisseur de l'implémentation de l'API pour générer automatiquement la table dans la base de données.

Il faut obligatoirement définir une des propriétés de la classe avec l'annotation @Id pour la déclarer comme étant la clé primaire de la table.

Cette annotation peut marquer soit le champ de la classe concernée soit le getter de la propriété. L'utilisation de l'un ou l'autre précise au gestionnaire s'il doit se baser sur les champs ou les getter pour déterminer les associations entre l'entité et les champs de la table. La clé primaire peut être constituée d'une seule propriété ou composées de plusieurs propriétés qui peuvent être de type primitif ou chaîne de caractères.

La clé primaire composée d'un seul champ peut être une propriété d'un type primitif, ou une chaîne de caractères (String).

La clé primaire peut être générée automatiquement en utilisant l'annotation @javax.persistence.GeneratedValue. Cette annotation possède plusieurs attributs :

Attributs	Rôle
strategy	Précise le type de générateur à utiliser : TABLE, SEQUENCE, IDENTITY ou AUTO. La valeur par défaut est AUTO
generator	Nom du générateur à utiliser

Exemple :

```
package com.jmdoudoux.test.jpa;

import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@Entity
public class Personne implements Serializable {
  @Id
  @GeneratedValue
  private int id;

  private String prenom;

  private String nom;

  private static final long serialVersionUID = 1L;

  public Personne() {
    super();
  }

  public int getId() {
    return this.id;
  }

  public void setId(int id) {
    this.id = id;
  }

  public String getPrenom() {
    return this.prenom;
  }

  public void setPrenom(String prenom) {
    this.prenom = prenom;
  }

  public String getNom() {
    return this.nom;
  }

  public void setNom(String nom) {
    this.nom = nom;
  }

}
```

Le type AUTO est le plus généralement utilisé : il laisse l'implémentation générer la valeur de la clé primaire.

Le type IDENTITY utilise un type de colonne spécial de la base de données.

Le type TABLE utilise une table dédiée qui stocke les clés des tables générées. L'utilisation de cette stratégie nécessite l'utilisation de l'annotation @javax.persistence.TableGenerator

L'annotation @TableGenerator possède plusieurs attributs :

Attributs	Rôle
name	Nom identifiant le TableGenerator : il devra être utilisé comme valeur dans l'attribut generator de l'annotation @Id
table	Nom de la table utilisé
catalog	Nom du catalogue utilisé
schema	Nom du schéma utilisé
pkColumnName	Nom de la colonne qui précise la clé primaire à générer
valueColumnName	Nom de la colonne qui contient la valeur de la clé primaire générer
pkColumnValue	
allocationSize	Valeur utilisée lors de l'incrémentation de la valeur de la clé primaire
uniqueConstraints	

Le type SEQUENCE utilise un mécanisme nommé séquence proposé par certaines bases de données notamment celles d'Oracle. L'utilisation de cette stratégie nécessite l'utilisation de l'annotation @javax.persistence.SequenceGenerator

L'annotation @SequenceTableGenerator possède plusieurs attributs :

Attributs	Rôle
name	Nom identifiant le SequenceTableGenerator : il devra être utilisé comme valeur dans l'attribut generator de l'annotation @Id
sequenceName	Nom de la séquence dans la base de données
initialValue	Valeur initiale de la séquence
allocationSize	Valeur utilisée lors l'incrémentation de la valeur de la séquence

L'annotation @SequenceGenerator s'utilise sur la classe de l'entité

Exemple :

```
@Entity
@Table(name="PERSONNE")
@SequenceGenerator(name="PERSONNE_SEQUENCE",
sequenceName="PERSONNE_SEQ")
public class Personne implements Serializable {
  @Id
  @GeneratedValue(strategy=GenerationType.SEQUENCE, generator="PERSONNE_SEQUENCE")
  private int id;
```

Le modèle de base de données relationnelle permet la définition d'une clé primaire composée de plusieurs colonnes. L'API Java Persistence propose deux façons de gérer ce cas de figure :

* L'annotation @javax.persistence.IdClass
* L'annotation @javax.persistence.EmbeddedId

L'annotation @IdClass s'utilise avec une classe qui va encapsuler les propriétés qui composent la clé primaire. Cette classe doit obligatoirement :

* Être sérialisable
* Posséder un constructeur sans argument
* Fournir une implémentation dédiée des méthodes equals() et hashCode()

Exemple : la clé primaire est composée des champs nom et prenom (exemple théorique qui présume que deux personnes ne peuvent avoir le même nom et prénom)

```
package com.jmdoudoux.test.jpa;

public class PersonnePK implements java.io.Serializable {

  private static final long serialVersionUID = 1L;

  private String nom;

  private String prenom;

  public PersonnePK() {
  }

  public PersonnePK(String nom, String prenom) {
    this.nom = nom;
    this.prenom = prenom;
  }

  public String getNom() {
    return this.nom;
  }

  public void setNom(String nom) {
    this.nom = nom;
  }

  public String getPrenom() {
    return prenom;
  }

  public void setPrenom(String prenom) {
    this.nom = prenom;
  }

  public boolean equals(Object obj) {
    boolean resultat = false;

    if (obj == this) {
      resultat = true;
    } else {
      if (!(obj instanceof PersonnePK)) {
        resultat = false;
      } else {
        PersonnePK autre = (PersonnePK) obj;
        if (!nom.equals(autre.nom)) {
          resultat = false;
        } else {
          if (prenom != autre.prenom) {
            resultat = false;
          } else {
            resultat = true;
          }
        }
      }
    }
    return resultat;
  }

  public int hashCode() {
```

```
    return (nom + prenom).hashCode();
  }
}
```

Il est nécessaire de définir la classe de la clé primaire dans le fichier de configuration persistence.xml

Exemple :

```
<?xml version="1.0"  encoding="UTF-8"?>
<persistence  xmlns="http://java.sun.com/xml/ns/persistence"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              version="1.0"  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
              http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
    <persistence-unit  name="MaBaseDeTestPU">
         <provider>oracle.toplink.essentials.PersistenceProvider</provider>
         <class>com.jmdoudoux.test.jpa.Personne</class>
         <class>com.jmdoudoux.test.jpa.PersonnePK</class>
    </persistence-unit>
</persistence>
```

L'annotation @IdClass possède un seul attribut :

Attributs	Rôle
Class	Classe qui encapsule la clé primaire composée

Il faut utiliser l'annotation@IdClass sur la classe de l'entité.

Il est nécessaire de marquer chacune des propriétés de l'entité qui compose la clé primaire avec l'annotation @Id. Ces propriétés doivent avoir le même nom dans l'entité et dans la classe qui encapsule la clé primaire.

Exemple :

```
package com.jmdoudoux.test.jpa;

import java.io.Serializable;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.IdClass;

@Entity
@IdClass(PersonnePK.class)
public class Personne implements Serializable {
  private String prenom;
  private String nom;
  private int taille;

  private static final long serialVersionUID = 1L;

  public Personne() {
    super();
  }

  @Id
  public String getPrenom() {
    return this.prenom;
  }

  public void setPrenom(String prenom) {
    this.prenom = prenom;
  }

  @Id
  public String getNom() {
    return this.nom;
```

```
  }

  public void setNom(String nom) {
    this.nom = nom;
  }

  public int getTaille() {
    return this.taille;
  }

  public void setTaille(int taille) {
    this.taille = taille;
  }
}
```

Remarque : il n'est pas possible de demander la génération automatique d'une clé primaire composée. Les valeurs de chacune des propriétés de la clé doivent être fournies explicitement.

La classe de la clé primaire est utilisée notamment lors des recherches.

Exemple :

```
PersonnePK clePersonne = new PersonnePK("nom1", "prenom1");
Personne personne = entityManager.find(Personne.class, clePersonne);
```

L'annotation @EmbeddedId s'utilise avec l'annotation @javax.persistence.Embeddable

Exemple :

```
package com.jmdoudoux.test.jpa;

import javax.persistence.Embeddable;

@Embeddable
public class PersonnePK implements java.io.Serializable {

  private static final long serialVersionUID = 1L;

  private String nom;

  private String prenom;

  public PersonnePK() {
  }

  public PersonnePK(String nom, String prenom) {
    this.nom = nom;
    this.prenom = prenom;
  }

  public String getNom() {
    return this.nom;
  }

  public void setNom(String nom) {
    this.nom = nom;
  }

  public String getPrenom() {
    return prenom;
  }

  public void setPrenom(String prenom) {
    this.nom = prenom;
  }

  public boolean equals(Object obj) {
    boolean resultat = false;
```

```
    if (obj == this) {
      resultat = true;
    } else {
      if (!(obj instanceof PersonnePK)) {
        resultat = false;
      } else {
        PersonnePK autre = (PersonnePK) obj;
        if (!nom.equals(autre.nom)) {
          resultat = false;
        } else {
          if (prenom != autre.prenom) {
            resultat = false;
          } else {
            resultat = true;
          }
        }
      }
    }
    return resultat;
  }

  public int hashCode() {
    return (nom + prenom).hashCode();
  }
}
```

Exemple :

```
package com.jmdoudoux.test.jpa;

import java.io.Serializable;

import javax.persistence.EmbeddedId;
import javax.persistence.Entity;

@Entity
public class Personne implements Serializable {
  @EmbeddedId
  private PersonnePK clePrimaire;
  private int taille;

  private static final long serialVersionUID = 1L;

  public Personne() {
    super();
  }

  public PersonnePK getClePrimaire() {
    return this.clePrimaire;
  }

  public void setNom(PersonnePK clePrimaire) {
    this.clePrimaire = clePrimaire;
  }

  public int getTaille() {
    return this.taille;
  }

  public void setTaille(int taille) {
    this.taille = taille;
  }
}
```

La classe qui encapsule la clé primaire est utilisée notamment dans les recherches

Exemple :

```
PersonnePK clePersonne = new PersonnePK("nom1", "prenom1");
```

```
Personne personne = entityManager.find(Personne.class, clePersonne);
```

L'annotation @AttributeOverrides est une collection d'attribut @AttributeOverride. Ces annotations permettent de ne pas avoir à utiliser l'annotation @Column dans la classe de la clé ou de modifier les attributs de cette annotation dans l'entité qui la met en oeuvre.

L'annotation @AttributeOverride possède plusieurs attributs :

Attributs	Rôle
name	Précise le nom de la propriété de la classe imbriquée
column	Précise la colonne de la table à associer à la propriété

Exemple :

```
package com.jmdoudoux.test.jpa;

import java.io.Serializable;

import javax.persistence.AttributeOverrides;
import javax.persistence.AttributeOverride;
import javax.persistence.EmbeddedId;
import javax.persistence.Entity;
import javax.persistence.Column;

@Entity
public class Personne4 implements Serializable {
  @EmbeddedId
  @AttributeOverrides({
    @AttributeOverride(name="nom", column=@Column(name="NOM") ),
    @AttributeOverride(name="prenom", column=@Column(name="PRENOM") )
    })
  private PersonnePK clePrimaire;
  private int taille;
...
```

Par défaut, toutes les propriétés sont mappées sur la colonne correspondante dans la table. L'annotation @javax.persistence.Transient permet d'indiquer au gestionnaire de persistance d'ignorer cette propriété.

L'annotation @javax.persistence.Basic représente la forme de mapping la plus simple. C'est aussi celle par défaut ce qui rend son utilisation optionnelle. Ce mapping concerne les types primitifs, les wrappers de type primitifs, les tableaux de ces types et les types java.math.BigInteger, java.math.BigDecimal, java.util.Date, java.util.Calendar, java.sql.Date, java.sql.Time et java.sql.Timestamp.

L'annotation @Basic possède plusieurs attributs :

Attributs	Rôle
fetch	Permet de préciser comment la propriété est chargée selon deux modes : • LAZY : la valeur est chargée uniquement lors de son utilisation • EAGER : la valeur est toujours chargée (valeur par défaut) Cette fonctionnalité permet de limiter la quantité de données obtenue par une requête
optionnal	Indique que la colonne est nullable

Généralement, cette annotation peut être omise sauf dans le cas ou le chargement de la propriété doit être de type LAZY.

Exemple :

```
package com.jmdoudoux.test.jpa;

import java.io.Serializable;

import javax.persistence.Basic;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@Entity
public class Personne implements Serializable {
  @Id
  @GeneratedValue
  private int id;

  @Basic(fetch=FetchType.LAZY, optional=false)
  private String prenom;

  private String nom;
...
```

L'annotation @javax.persistence.Temporal permet de fournir des informations complémentaires sur la façon dont les propriétés encapsulant des données temporelles (Date et Calendar) sont associées aux colonnes dans table (date, time ou timestamp). La valeur par défaut est timestamp.

Exemple :

```
package com.jmdoudoux.test.jpa;

import java.io.Serializable;
import java.util.Date;

import javax.persistence.Basic;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

@Entity
public class Personne implements Serializable {
  @Id
  @GeneratedValue
  private int id;

  @Basic(fetch = FetchType.LAZY, optional = false)
  private String prenom;

  private String nom;

  @Temporal(TemporalType.TIME)
  private Date heureNaissance;

  private static final long serialVersionUID = 1L;

  public Personne() {
    super();
  }

  public int getId() {
    return this.id;
  }

  public void setId(int id) {
    this.id = id;
  }

  public Date getHeureNaissance() {
    return heureNaissance;
```

```
  }

  public void setTimeCreated(Date heureNaissance) {
    this.heureNaissance = heureNaissance;
  }
...
```

43.2.2. Le mapping de propriété complexe

L'API Java persistence permet de mapper des colonnes qui concernent des données de type plus complexe que les types de base tel que les champs blob ou clob ou des objets.

L'annotation @javax.persistence.Lob permet mapper une propriété sur une colonne de type Blob ou Clob selon le type de la propriété :

- Blob pour les tableaux de byte ou Byte ou les objets sérializables
- Clob pour les chaînes de caractères et les tableaux de caractères char ou Char

Fréquemment ce type de propriété est chargé de façon LAZY.

Exemple :

```
package com.jmdoudoux.test.jpa;

import java.io.Serializable;

import javax.persistence.Basic;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Lob;

import com.sun.imageio.plugins.jpeg.JPEG;

@Entity
public class Personne implements Serializable {
  @Id
  @GeneratedValue
  private int id;

  @Basic(fetch = FetchType.LAZY, optional = false)
  private String prenom;

  private String nom;

  @Lob
  @Basic(fetch = FetchType.LAZY)
  private JPEG photo;

  private static final long serialVersionUID = 1L;

  public Personne() {
    super();
  }

  public int getId() {
    return this.id;
  }

  public void setId(int id) {
    this.id = id;
  }

  public JPEG getPhoto() {
    return photo;
```

```
    }

    public void setPhoto(JPEG photo) {
      this.photo = photo;
    }

    ...
```

L'annotation @javax.persistence.Enumerated permet d'associer une propriété de type énumération à une colonne de la table sous la forme d'un numérique ou d'une chaîne de caractères.

Cette forme est précisée en paramètre de l'annotation grâce à l'énumération EnumType qui peut avoir comme valeur EnumType.ORDINAL (valeur par défaut) ou EnumType.STRING.

Exemple : énumération des genres d'une personne

```
package com.jmdoudoux.test.jpa;

public enum Genre {
   HOMME,
   FEMME,
   INCONNU
}
```

Exemple :

```
package com.jmdoudoux.test.jpa;

import java.io.Serializable;

import javax.persistence.Basic;
import javax.persistence.Entity;
import javax.persistence.EnumType;
import javax.persistence.Enumerated;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@Entity
public class Personne implements Serializable {
  @Id
  @GeneratedValue
  private int id;

  @Basic(fetch = FetchType.LAZY, optional = false)
  private String prenom;

  private String nom;

  @Enumerated(EnumType.STRING)
  private Genre genre;

  private static final long serialVersionUID = 1L;

  public Personne() {
    super();
  }

  public int getId() {
    return this.id;
  }

  public void setId(int id) {
    this.id = id;
  }

  public Genre getGenre() {
    return genre;
  }
```

```
  public void setPhoto(Genre genre) {
    this.genre = genre;
  }

  ...
```

43.2.3. Le mapping d'une entité sur plusieurs tables

Le modèle objet et le modèle relationnel correspondant ne correspondent pas toujours car les critères de conception ne sont pas forcement les même. Ainsi, il est courant d'avoir une entité qui mappe des colonnes de plusieurs tables.

Exemple : création de la table adresse utilisée dans cette section

```
ij> create table ADRESSE
(
ID_ADRESSE integer primary key not null,
RUE varchar(250) not null,
CODEPOSTAL varchar(7) not null,
VILLE varchar(250) not null
);
0 lignes insérées/mises à jour/supprimées
ij> INSERT INTO ADRESSE VALUES (1,'rue1','11111','ville1'), (2,'rue2','22222','v
ille2'), (3,'rue3','33333','ville3');
3 lignes insérées/mises à jour/supprimées
ij> select * from adresse;
ID_ADRESSE |RUE
                                                            |CODEPO&|VILLE


--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
-------------------------------------
1          |rue1
                                                            |11111  |ville1


2          |rue2
                                                            |22222  |ville2


3          |rue3
                                                            |33333  |ville3



3 lignes sélectionnées
ij>
```

L'annotation @javax.persistence.SecondaryTable permet de préciser qu'une autre table sera utilisée dans le mapping.

Pour utiliser cette fonctionnalité, la seconde table doit posséder une jointure entre sa clé primaire et une ou plusieurs colonnes de la première table.

L'annotation @SecondaryTable possède plusieurs attributs :

Attribut	Rôle
Name	Nom de la table
Catalogue	Nom du catalogue
Schema	Nom du schéma
pkJoinsColumns	

	Collection des clés primaire de la jointure sous la forme d'annotations de type @PrimaryKeyJoinColumn
uniqueConstraints	

L'annotation @PrimaryKeyJoinColumn permet de préciser une colonne qui compose la clé primaire de la seconde table et entre dans la jointure avec la première table. Elle possède plusieurs attributs :

Attribut	Rôle
name	Nom de la colonne
referencedColumnName	Nom de la colonne dans la première table (obligatoire si les noms de colonnes sont différents entre les deux tables)
columnDefinition	

Il est nécessaire pour chaque propriété de l'entité qui est mappée sur la seconde table de renseigner le nom de la table dans l'attribut table de l'annotation @Column

Exemple : la classe PersonneAdresse

```
package com.jmdoudoux.test.jpa;

import java.io.Serializable;

import javax.persistence.Basic;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.PrimaryKeyJoinColumn;
import javax.persistence.SecondaryTable;
import javax.persistence.Table;

@Entity
@Table(name="PERSONNE")
@SecondaryTable(name="ADRESSE",
pkJoinColumns={
@PrimaryKeyJoinColumn(name="ID_ADRESSE")})
public class PersonneAdresse implements Serializable {
  @Id
  @GeneratedValue
  private int id;

  @Basic(fetch = FetchType.LAZY, optional = false)
  private String prenom;

  private String nom;

  @Column(name="RUE", table="ADRESSE")
  private String rue;

  @Column(name="CODEPOSTAL", table="ADRESSE")
  private String codePostal;

  @Column(name="VILLE", table="ADRESSE")
  private String ville;

  private static final long serialVersionUID = 1L;

  public PersonneAdresse() {
    super();
  }

  public int getId() {
    return this.id;
```

```
  }

  public void setId(int id) {
    this.id = id;
  }

  public String getPrenom() {
    return this.prenom;
  }

  public void setPrenom(String prenom) {
    this.prenom = prenom;
  }

  public String getNom() {
    return this.nom;
  }

  public void setNom(String nom) {
    this.nom = nom;
  }

  public String getCodePostal() {
    return codePostal;
  }

  public void setCodePostal(String codePostal) {
    this.codePostal = codePostal;
  }

  public String getRue() {
    return rue;
  }

  public void setRue(String rue) {
    this.rue = rue;
  }

  public String getVille() {
    return ville;
  }

  public void setVille(String ville) {
    this.ville = ville;
  }

}
```

Résultat :

```
nom prenom=nom1 prenom1
adresse=rue1, 11111 ville1
```

Si le mapping d'une entité met en oeuvre plus de deux tables, il faut utiliser l'annotation @javax.persistence.SecondaryTables qui est une collection d'annotation @SecondaryTable

Exemple :

```
package com.jmdoudoux.test.jpa;

import java.io.Serializable;

import javax.persistence.Basic;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.PrimaryKeyJoinColumn;
import javax.persistence.SecondaryTable;
```

```
import javax.persistence.SecondaryTables;
import javax.persistence.Table;

@Entity
@Table(name="PERSONNE")
@SecondaryTables({
  @SecondaryTable(name="ADRESSE",
     pkJoinColumns={@PrimaryKeyJoinColumn(name="ID_ADRESSE")}),
  @SecondaryTable(name="INFO_PERS",
     pkJoinColumns={@PrimaryKeyJoinColumn (name="ID_INFO_PERS")})
})
public class PersonneAdresse implements Serializable {
...
```

43.2.4. L'utilisation d'objets embarqués dans les entités

L'API Java Persistence permet d'utiliser dans les entités des objets Java qui ne sont pas des entités mais qui sont agrégés dans l'entité et dont les propriétés seront mappées sur les colonnes correspondantes dans la table.

La mise en oeuvre de cette fonctionnalité est similaire à celle utilisée avec l'annotation @EmbeddedId pour les clés primaires composées.

La classe embarquée est un simple POJO qui doit être marquée avec l'annotation @javax.persistence.Embeddable

Exemple :

```
package com.jmdoudoux.test.jpa;

import java.io.Serializable;

import javax.persistence.Column;
import javax.persistence.Embeddable;

@Embeddable
public class Adresse implements Serializable{

  private static final long serialVersionUID = 1L;

  @Column(name="RUE", table="ADRESSE")
  private String rue;

  @Column(name="CODEPOSTAL", table="ADRESSE")
  private String codePostal;

  @Column(name="VILLE", table="ADRESSE")
  private String ville;

  public String getCodePostal() {
    return codePostal;
  }

  public void setCodePostal(String codePostal) {
    this.codePostal = codePostal;
  }

  public String getRue() {
    return rue;
  }

  public void setRue(String rue) {
    this.rue = rue;
  }

  public String getVille() {
    return ville;
  }
```

```
  public void setVille(String ville) {
    this.ville = ville;
  }

}
```

Les propriétés de cette classe peuvent être marquées avec l'annotation @Column au besoin.

Dans l'entité, il faut utiliser l'annotation @javax.persistence.Embedded sur la propriété du type de la classe embarquée.

Exemple :

```
package com.jmdoudoux.test.jpa;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class TestJPA3 {

  public static void main(String[] argv) {
    EntityManagerFactory emf = Persistence
        .createEntityManagerFactory("MaBaseDeTestPU");
    EntityManager em = emf.createEntityManager();
    PersonneAdresse2 personneAdresse = em.find(PersonneAdresse2.class, 1);
    System.out.println("nom prenom="
        + personneAdresse.getNom()
        + " "
        + personneAdresse.getPrenom());
    System.out.println("adresse="
        + personneAdresse.getAdresse().getRue()
        + ", "
        + personneAdresse.getAdresse().getCodePostal()
        + " "
        + personneAdresse.getAdresse().getVille());
    em.close();
    emf.close();
  }
}
```

L'annotation @AttributesOverride peut être utilisé pour adapter au contexte le mapping des propriétés de l'objet embarqué.

Exemple :

```
nom prenom=nom1 prenom1
adresse=rue1, 11111 ville1
```

Si l'annotation @Embedded n'est pas utilisée alors le gestionnaire de persistance va mapper la propriété sur le champ correspondant sous sa forme sérialisée. Avec l'annotation @Embedded chaque propriété de l'objet embarqué est mappé sur la colonne correspondante de la table.

43.3. Le fichier de configuration du mapping

Il est aussi possible de définir le mapping dans un fichier de mapping nommé par défaut orm.xml stocké dans le répertoire META-INF.

Ce fichier orm.xml est un fichier au format xml. L'élément racine est le tag <entity-mappings>.

Pour chaque entité, il faut utiliser un tag fils <entity>. Ce tag possède deux attributs :

- Class qui permet préciser le nom pleinement qualifié de la classe de l'entité
- Access qui permet de préciser le type d'accès aux données (PROPERTY pour un accès via les getter/setter ou FIELD pour un accès via les champs).

La déclaration de la clé primaire se fait dans un tag <id> fils d'un tag <attributes>. Ce tag <id> possède un attribut nommé name qui permet de préciser le nom du champ qui est la clé primaire.

La suite de ce chapitre sera développée dans une version future de ce document

Le fichier de mapping peut aussi avoir un nom arbitraire mais dans ce cas, il devra être précisé avec le tag <mapping-file> dans le fichier de configuration persistence.xml

43.4. L'utilisation du bean entité

Comme c'est un POJO, il est possible d'ajouter des méthodes à la classe mais il est cependant conseillé de maintenir le rôle du bean entity au transfert de données : il faut éviter de lui ajouter des méthodes métiers mais il est possible de définir des méthodes de validation des données qu'il encapsule.

La mise en oeuvre de POJO permet de les utiliser directement lors d'échanges entre le client et le serveur car ils peuvent être sérialisés comme tout objet de base Java.

Les POJO ne servent qu'à définir le mapping et encapsuler des données. L'instanciation d'une entité n'a aucune conséquence sur la table de la base de données mappée avec l'objet.

Toutes les actions de persistance sur ces objets sont réalisées grâce à un objet dédié de l'API : l'EntityManager.

43.4.1. L'utilisation du bean entité

Un contexte de persistance (persistence context) est un ensemble d'entités géré par un EntityManager.

Les entités peuvent ainsi être de deux types :

- Gérée (Managed) :
- Non gérée (Unmanaged) :

Lorsqu'un contexte de persistance est fermé, toutes les entités du contexte deviennent non gérées.

Il existe deux types de contexte de persistance :

- Transaction-scoped : le contexte est ouvert pendant toute la durée d'une transaction. La fermeture de la transaction entraîne la fermeture du contexte. Ce type de contexte n'est utilisable que dans le cadre de l'utilisation dans un conteneur qui va assurer la prise en charge de la transaction
- Extended persitence : le contexte reste ouvert après la fermeture de la transaction

43.4.2. L'EntityManager

Les interactions entre la base de données et les beans entité sont assurées par un objet de type javax.persistence.EntityManager : il permet de lire et rechercher des données mais aussi de les mettre à jour (ajout, modification, suppression). L'EntityManager est donc au coeur de toutes les actions de persistance.

Les bean entité étant de simple POJO, leur instanciation se fait comme pour tout autre objet Java. Les données de cette instance ne sont rendues persistantes que par une action explicite demandée à l'EntityManager sur le bean entité.

L'EntityManager assure aussi les interactions avec un éventuel gestionnaire de transactions.

Un EntityManager gère un ensemble défini de beans entité nommé persistence unit. La définition d'un persistence unit est assurée dans un fichier de description nommé persistence.xml.

43.4.2.1. L'obtention d'une instance de la classe EntityManager

Lors d'une utilisation dans un conteneur Java EE, il est possible d'obtenir un objet de type EntityManager en utilisant l'injection de dépendance pour l'objet lui-même ou obtenir une fabrique de type EntityManagerFactory qui sera capable de créer l'objet.

Dans un environnement Java SE, comme par exemple dans Tomcat ou dans une application de type client lourd, l'instanciation d'un objet de type EntityManager doit être codée.

Sous Java SE, pour obtenir une instance de type EntityManager, il faut utiliser une fabrique de type EntityManagerFactory. Cette fabrique propose la méthode createEntityManager() pour obtenir une instance.

Pour obtenir une instance de la fabrique, il utiliser la méthode statique createEntityManagerFactory() de la classe javax.persistence.Persistence qui attend en paramètre le nom de l'unité de persistence à utiliser. Elle va rechercher le fichier persistence.xml dans le classpath et recherche dans ce fichier l'unité de persistance dont le nom est fourni.

Il faut utiliser la méthode close() de la fabrique une fois que cette dernière n'a plus d'utilité pour libérer les ressources.

Sous Java EE, il est préférable d'utiliser l'injection de dépendance pour obtenir une fabrique ou un contexte de persistance.

L'annotation @javax.persistence.PersistenceUnit sur un champ de type EntityManagerFactory permet d'injecter une fabrique. Cette annotation possède un attribut unitName qui précise le nom de l'unité de persistance.

Exemple :

```
@PersistenceUnit(unitName="MaBaseDeTestPU")
private EntityManagerFactory factory;
```

Il est alors possible d'utiliser la fabrique pour obtenir un objet de type EntityManager qui encapsule un contexte de persistence de type extended. Pour associer ce contexte à la transaction courante, il faut utiliser la méthode joinTransaction().

La méthode close() est automatiquement appelée par le conteneur : il ne faut pas utiliser cette méthode dans un conteneur sinon une exception de type IllegalStateException est levée.

L'annotation @javax.persistence.PersistenceContext sur un champ de type EntityManager permet d'injecter un contexte de persistance. Cette annotation possède un attribut unitName qui précise le nom de l'unité de persistance.

Exemple :

```
@PersistenceContext(unitName="MaBaseDeTestPU")
private EntityManager entityManager;
```

43.4.2.2. L'utilisation de la classe EntityManager

La méthode contains() de l'EntityManager permet de savoir si une instance fournie en paramètre est gérée par le contexte. Dans ce cas, elle renvoie true, sinon elle renvoie false.

La méthode clear() de l'EntityManager permet de détacher toutes les entités gérées par le contexte. Dans ce cas, toutes les modifications apportées aux entités sont perdues : il est préférable d'appeler la méthode flush() avant la méthode clear() afin de rendre persistante toutes les modifications.

L'appel des méthodes de mise à jour persist(), merge() et remove() ne réalise pas d'actions immédiates dans la base de données sous jacente. L'exécution de ces actions est à la discrétion de l'EntityManager selon le FlushModeType (AUTO ou COMMIT).

Dans le mode AUTO, les mises à jour sont reportées dans la base de données avant chaque requête. Dans le mode COMMIT, les mises à jour sont reportées dans la base de données lors du commit de la transaction.

Le mode COMMIT est plus performant car il limite les échanges avec la base de données.

Il est possible de forcer l'enregistrement des mises à jour dans la base de données en utilisant la méthode flush() de l'EntityManager.

43.4.2.3. L'utilisation de la classe EntityManager pour la création d'une occurrence

Pour insérer une nouvelle entité dans la base de données, il faut :

- Instancier une occurrence de la classe de l'entité
- Initialiser les propriétés de l'entité
- Définir les relations de l'entité avec d'autres entités au besoin
- Utiliser la méthode persist() de l'EntityManager en passant en paramètre l'entité

Exemple :

```
package com.jmdoudoux.test.jpa;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;

public class TestJPA4 {
  public static void main(String[]argv) {
    EntityManagerFactory emf = Persistence.createEntityManagerFactory("MaBaseDeTestPU");
    EntityManager em = emf.createEntityManager();
    EntityTransaction transac = em.getTransaction();
    transac.begin();
    Personne nouvellePersonne = new Personne();
    nouvellePersonne.setId(4);
    nouvellePersonne.setNom("nom4");
    nouvellePersonne.setPrenom("prenom4");
    em.persist(nouvellePersonne);
    transac.commit();

    em.close();
    emf.close();
  }
}
```

Remarque : l'exemple ci-dessous utilise JPA dans un environnement qui ne propose aucune fonctionnalité pour assurer les transactions (Java SE) : il est donc nécessaire de créer et gérer manuellement une transaction afin d'assurer la persistance des données.

43.4.2.4. L'utilisation de la classe EntityManager pour rechercher des occurrences

Pour effectuer des recherches de données, l'EntityManager propose deux mécanismes :

- La recherche à partir de la clé primaire

* La recherche à partir d'une requête utilisant une syntaxe dédiée

Pour la recherche par clé primaire, la classe EntityManager possède les méthodes find() et getReference() qui attendent toutes les deux en paramètres un objet de type Class représentant la classe de l'entité et un objet qui contient la valeur de la clé primaire.

La méthode find() renvoie null si l'occurrence n'est pas trouvée dans la base de données.

Exemple :

```
package com.jmdoudoux.test.jpa;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class TestJPA5 {
  public static void main(String[] argv) {
    EntityManagerFactory emf = Persistence
        .createEntityManagerFactory("MaBaseDeTestPU");
    EntityManager em = emf.createEntityManager();

    Personne personne = em.find(Personne.class, 4);
    if (personne != null) {
      System.out.println("Personne.nom=" + personne.getNom());
    }
    em.close();
    emf.close();
  }
}
```

La méthode getReference() lève une exception de type javax.persistence.EntityNotFoundException si l'occurrence n'est pas trouvée dans la base de données.

Exemple :

```
package com.jmdoudoux.test.jpa;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityNotFoundException;
import javax.persistence.Persistence;

public class TestJPA6 {
  public static void main(String[] argv) {
    EntityManagerFactory emf = Persistence
        .createEntityManagerFactory("MaBaseDeTestPU");
    EntityManager em = emf.createEntityManager();

    try {
    Personne personne = em.getReference(Personne.class, 5);
      System.out.println("Personne.nom=" + personne.getNom());
    } catch (EntityNotFoundException e) {
      System.out.println("personne non trouvée");
    }
    em.close();
    emf.close();
  }
}
```

43.4.2.5. L'utilisation de la classe EntityManager pour rechercher des données par requête

La recherche par requête repose sur des méthodes dédiées de la classe EntityManager (createQuery(), createNamedQuery() et createNativeQuery()) et sur un langage de requête spécifique nommé EJB QL.

L'objet Query encapsule et permet d'obtenir les résultats de son exécution. La méthode getSingleResult() permet d'obtenir un objet unique retourné par la requête.

Exemple :

```
package com.jmdoudoux.test.jpa;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.Query;

public class TestJPA7 {
  public static void main(String[] argv) {
    EntityManagerFactory emf = Persistence
        .createEntityManagerFactory("MaBaseDeTestPU");
    EntityManager em = emf.createEntityManager();

    Query query = em.createQuery("select p from Personne p where p.nom='nom2'");
    Personne personne = (Personne) query.getSingleResult();
    if (personne == null) {
      System.out.println("Personne non trouvée");
    } else {
      System.out.println("Personne.nom=" + personne.getNom());
    }

    em.close();
    emf.close();
  }
}
```

La méthode getResultList() renvoie une collection qui contient les éventuelles occurrences retournées par la requête.

Exemple :

```
package com.jmdoudoux.test.jpa;

import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.Query;

public class TestJPA8 {
  public static void main(String[] argv) {
    EntityManagerFactory emf = Persistence
        .createEntityManagerFactory("MaBaseDeTestPU");
    EntityManager em = emf.createEntityManager();

    Query query = em.createQuery("select p.nom from Personne p where p.id > 2");
    List noms = query.getResultList();
    for (Object nom : noms) {
      System.out.println("nom = "+nom);
    }

    em.close();
    emf.close();
  }
}
```

L'objet Query gère aussi des paramètres nommés dans la requête. Le nom de chaque paramètre est préfixé par « : » dans la requête. La méthode setParameter() permet de fournir une valeur à chaque paramètre.

Exemple :

```
package com.jmdoudoux.test.jpa;
```

```
import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.Query;

public class TestJPA9 {
  public static void main(String[] argv) {
    EntityManagerFactory emf = Persistence
        .createEntityManagerFactory("MaBaseDeTestPU");
    EntityManager em = emf.createEntityManager();

    Query query = em.createQuery("select p.nom from Personne p where p.id > :id");
    query.setParameter("id", 1);
    List noms = query.getResultList();
    for (Object nom : noms) {
      System.out.println("nom = "+nom);
    }

    em.close();
    emf.close();
  }
}
```

43.4.2.6. L'utilisation de la classe EntityManager pour modifier une occurrence

Pour modifier une entité existante dans la base de données, il faut :

- Obtenir une instance de l'entité à modifier (par recherche sur la clé primaire ou l'exécution d'une requête)
- Modifier les propriétés de l'entité
- Selon le mode de synchronisation des données de l'EntityManager, il peut être nécessaire d'appeler la méthode flush() explicitement

Exemple :

```
package com.jmdoudoux.test.jpa;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;
import javax.persistence.Query;

public class TestJPA10 {
  public static void main(String[] argv) {
    EntityManagerFactory emf = Persistence
        .createEntityManagerFactory("MaBaseDeTestPU");
    EntityManager em = emf.createEntityManager();

    EntityTransaction transac = em.getTransaction();
    transac.begin();

    Query query = em.createQuery("select p from Personne p where p.nom='nom2'");
    Personne personne = (Personne) query.getSingleResult();
    if (personne == null) {
      System.out.println("Personne non trouvée");
    } else {
      System.out.println("Personne.prenom=" + personne.getPrenom());

      personne.setPrenom("prenom2 modifie");
      em.flush();

      personne = (Personne) query.getSingleResult();
      System.out.println("Personne.prenom=" + personne.getPrenom());
    }

    transac.commit();
```

```
    em.close();
    emf.close();
  }
}
```

43.4.2.7. L'utilisation de la classe EntityManager pour fusionner des données

L'EntityManager propose la méthode merge() pour fusionner les données d'une entité non gérée avec la base de données. Ceci est particulièrement utile notamment lorsque l'entité est sérialisée pour être envoyée au client : dans ce cas, l'entité n'est plus gérée par le contexte. Lorsque le client renvoie l'entité modifiée, il faut synchroniser les données qu'elle contient avec celle de la base de données. C'est le rôle de la méthode merge().

Exemple :

```
package com.jmdoudoux.test.jpa;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;
import javax.persistence.Query;

public class TestJPA11 {
  public static void main(String[] argv) {
    EntityManagerFactory emf = Persistence
        .createEntityManagerFactory("MaBaseDeTestPU");
    EntityManager em = emf.createEntityManager();

    EntityTransaction transac = em.getTransaction();
    transac.begin();

    Query query = em.createQuery("select p from Personne p where p.nom='nom2'");
    Personne personne = (Personne) query.getSingleResult();
    if (personne == null) {
      System.out.println("Personne non trouvée");
    } else {
      System.out.println("Personne.prenom=" + personne.getPrenom());

      Personne pers = new Personne();
      pers.setId(personne.getId());
      pers.setNom(personne.getNom());
      pers.setPrenom("prenom2 REmodifie");

      em.merge(pers);

      personne = (Personne) query.getSingleResult();
      System.out.println("Personne.prenom=" + personne.getPrenom());
    }

    transac.commit();

    em.close();
    emf.close();
  }
}
```

La méthode merge() renvoie une instance gérée de l'entité.

43.4.2.8. L'utilisation de la classe EntityManager pour supprimer une occurrence

Pour supprimer une entité existante dans la base de données, il faut :

* Obtenir une instance de l'entité à supprimer (par recherche sur la clé primaire ou l'exécution d'une requête)
* Appeler la méthode remove() de l'EntityManager en lui passant en paramètre l'instance de l'entité

Exemple :

```
package com.jmdoudoux.test.jpa;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;

public class TestJPA12 {
  public static void main(String[] argv) {
    EntityManagerFactory emf = Persistence
        .createEntityManagerFactory("MaBaseDeTestPU");
    EntityManager em = emf.createEntityManager();

    EntityTransaction transac = em.getTransaction();
    transac.begin();

    Personne personne = em.find(Personne.class, 4);
    if (personne == null) {
      System.out.println("Personne non trouvée");
    } else {
      em.remove(personne);
    }

    transac.commit();

    em.close();
    emf.close();
  }
}
```

La seule façon d'annuler une suppression est de recréer l'entité en utilisant la méthode persist().

43.4.2.9. L'utilisation de la classe EntityManager pour rafraîchir les données d'une occurrence

La méthode refresh() de l'EntityManager permet de rafraîchir les données de l'entité avec celles contenues dans la base de données.

Exemple :

```
package com.jmdoudoux.test.jpa;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;

public class TestJPA13 {
  public static void main(String[] argv) {
    EntityManagerFactory emf = Persistence
        .createEntityManagerFactory("MaBaseDeTestPU");
    EntityManager em = emf.createEntityManager();

    EntityTransaction transac = em.getTransaction();
    transac.begin();

    Personne personne = em.find(Personne.class, 4);
    if (personne == null) {
      System.out.println("Personne non trouvée");
    } else {
      em.refresh(personne);
    }
```

```
        transac.commit();

        em.close();
        emf.close();
    }
}
```

La méthode refresh() peut lever une exception de type EntityNotFoundException si l'occurrence correspondante dans la base de données n'existe plus.

43.5. Le fichier persistence.xml

Ce fichier persistence.xml contient la configuration de base pour le mapping notamment en fournissant les informations sur la connexion à la base de données à utiliser.

Le fichier persistence.xml doit être stocké dans le répertoire META-INF

La racine du document XML du fichier persistence.xml est le tag <persistence>.

Il contient un ou plusieurs tags <persistence-unit> qui va contenir les paramètres d'un persistence unit. Ce tag possède deux attributs : name (obligatoire) qui précise le nom de l'unité et qui servira à y faire référence et transaction-type (optionnel) qui précise le type de transaction utilisée (ceci dépend de l'environnement d'exécution : Java SE ou Java EE).

Le tag <persistence-unit> peut avoir les tags fils suivants :

Tag	Rôle
<description>	Description purement informative de l'unité de persistance(optionnel)
<provider>	Nom pleinement qualifié d'une classe de type javax.persistence. PersistenceProvider (optionnel). Généralement fournie par le fournisseur de l'implémentation de l'API : une utilisation de ce tag n'est requise que pour des besoins spécifiques
<jta-data-source>	Nom JNDI de la DataSource utilisée dans un environnement avec support de JTA (optionnel)
<non-jta-data-source>	Nom JNDI de la DataSource utilisée dans un environnement sans support de JTA (optionnel)
<mapping-file>	Précise un fichier de mapping supplémentaire (optionnel)
<jar-file>	Précise un fichier jar qui contient des entités à inclure dans l'unité de persistance : le chemin précisé est relatif par rapport au fichier persistence.xml (optionnel)
<class>	Précise une classe d'une entité qui sera incluse dans l'unité de persistence (optionnel)
<properties>	Fournir des paramètres spécifiques au fournisseur. Comme Java SE ne propose pas de serveur JNDI, c'est fréquemment via ce tag que les informations concernant la source de données sont définis (optionnel)
<exclude-unlisted-classes>	Inhibition de la recherche automatique des classes des entités (optionnel)

L'ensemble des classes des entités qui compose l'unité de persistance peut être spécifié explicitement dans le fichier persistence.xml ou déterminé dynamiquement à l'exécution par recherche de toutes les classes possédant une annotation @javax.persistence.Entity.

Par défaut, la liste de classes explicite est complétée par la liste des classes issue de la recherche dynamique. Pour empêcher la recherche dynamique, il faut utiliser le tag <exclude-unlisted-classes>. Sous Java SE, il est recommandé de préciser explicitement la liste de classes.

Chaque unité de persistance ne peut être liée qu'à une seule source de données.

Exemple :

```
<?xml version="1.0"  encoding="UTF-8"?>
<persistence  xmlns="http://java.sun.com/xml/ns/persistence"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              version="1.0"  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
              http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
  <persistence-unit  name="MaBaseDeTestPU">
    <provider>oracle.toplink.essentials.PersistenceProvider</provider>
    <class>com.jmdoudoux.test.jpa.Adresse</class>
    <class>com.jmdoudoux.test.jpa.Personne</class>
    <class>com.jmdoudoux.test.jpa.PersonneAdresse</class>
    <class>com.jmdoudoux.test.jpa.PersonneAdresse2</class>
    <class>com.jmdoudoux.test.jpa.PersonnePK</class>
    <properties>
      <property name="toplink.jdbc.driver"
              value="org.apache.derby.jdbc.EmbeddedDriver"/>
      <property name="toplink.jdbc.url"
              value="jdbc:derby:C:/Program  Files/Java/jdk1.6.0/db/MaBaseDeTest"/>
      <property name="toplink.jdbc.user" value="APP"/>
      <property name="toplink.jdbc.password" value=""/>
      <property name="toplink.logging.level" value="INFO"/>
    </properties>
  </persistence-unit>
</persistence>
```

43.6. La gestion des transactions hors Java EE

Le conteneur Java EE propose un support des transactions grâce à l'API JTA : c'est la façon standard de gérer les transactions par le conteneur.

Hors d'un tel conteneur, par exemple dans une application Java SE, les transactions ne sont pas supportées.

Dans un tel contexte, l'API Java Persistence propose une gestion des transactions grâce à l'interface EntityTransaction.

Cette interface propose plusieurs méthodes :

Méthode	Rôle
void begin()	Débuter la transaction
void commit()	Valider la transaction
void roolback()	Annuler la transaction
boolean isActive()	Déterminer si la transaction est active

Pour obtenir une instance de la transaction, il faut utiliser la méthode getTransaction() de l'EntityManager.

La méthode begin() lève une exception de type IllegalStateException si une transaction est déjà active.

Les méthodes commit() et rollback() lèvent une exception de type IllegalStateException si aucune transaction n'est active.

Exemple :

```
package com.jmdoudoux.test.jpa;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;
```

```
public class TestJPA12 {
  public static void main(String[] argv) {
    EntityManagerFactory emf = Persistence
        .createEntityManagerFactory("MaBaseDeTestPU");
    EntityManager em = emf.createEntityManager();

    EntityTransaction transac = em.getTransaction();
    transac.begin();

    Personne personne = em.find(Personne.class, 4);
    if (personne == null) {
      System.out.println("Personne non trouvée");
    } else {
      em.remove(personne);
    }

    transac.commit();

    em.close();
    emf.close();
  }
}
```

43.7. La gestion des relations entre tables dans le mapping

Dans le modèle des bases de données relationnelles, les tables peuvent être liées entre elles grâce à des relations.

Ces relations sont transposées dans les liaisons que peuvent les différentes entités correspondantes.

Les relations peuvent avoir différentes cardinalités :

- 1-1 (one-to-one)
- 1-n (one-to-many)
- n-1 (many-to-one)
- n-n (many-to-many)

Chacune de ces relations peut être unidirectionnelle ou bidirectionnelle sauf one-to-many et many-to-one qui sont par définition mutuellement bidirectionnelles.

La suite de ce chapitre sera développée dans une version future de ce document

43.8. Les callbacks d'événements

L'API Java Persistence permet de définir des callbacks qui seront appelés sur certains événements. Ces callbacks doivent être annotés avec une des annotations définies par JPA

@javax.persistence.PrePersist
@javax.persistence.PostPersist
@javax.persistence.PostLoad
@javax.persistence.PreUpdate
@javax.persistence.PostUpdate

@javax.persistence.PreRemove
@javax.persistence.PostRemove

La suite de ce chapitre sera développée dans une version future de ce document

Partie 6 : La machine virtuelle Java (JVM)

Cette partie concerne la machine virtuelle Java ou JVM (Java Virtual Machine). La JVM est un des éléments les plus important de la plate-forme Java : une bonne compréhension de son fonctionnement et de certains des concepts qu'elle met en oeuvre est très important pour obtenir les meilleures performances avec certaines applications.

Cette partie regroupe plusieurs chapitres :

- La JVM (Java Virtual Machine) : ce chapitre détaille les différents éléments et concepts qui sont mis en oeuvre dans la JVM.
- La gestion de la mémoire : ce chapitre détaille la gestion de la mémoire dans la JVM et notamment les concepts et le paramétrage du ramasse miettes.
- La décompilation et l'obfuscation : ce chapitre présente la décompilation qui permet de transformer du byte code en code source et l'obfuscation qui est l'opération permettant de limiter cette transformation.
- Terracotta : Ce chapitre détaille les possibilités de l'outil open source Terracotta qui permet de mettre en cluster des JVM

44. La JVM (Java Virtual Machine)

La machine virtuelle Java ou JVM (Java Virtual Machine) est un environnement d'exécution pour applications Java.

C'est un des éléments les plus importants de la plate-forme Java. Elle assure l'indépendance du matériel et du système d'exploitation lors de l'exécution des applications Java. Une application Java ne s'exécute pas directement dans le système d'exploitation mais dans une machine virtuelle qui s'exécute dans le système d'exploitation et propose une couche d'abstraction entre l'application Java et ce système.

La machine virtuelle permet notamment :

- l'interprétation du byte code
- l'interaction avec le système d'exploitation
- la gestion de sa mémoire grâce au ramasse miettes

Son mode de fonctionnement est relativement similaire à celui d'un ordinateur : elle exécute des instructions qui manipulent différentes zones de mémoire dédiées de la JVM.

Une application Java ne fait pas d'appels directs au système d'exploitation (sauf en cas d'utilisation de JNI) : elle n'utilise que les API qui sont pour une large part écrites en Java sauf quelques unes qui sont natives. Ceci permet à Java de rendre les applications indépendantes de l'environnement d'exécution.

La machine virtuelle ne connaît pas le langage Java : elle ne connaît que le byte code qui est issu de la compilation de code source écrit Java.

Les spécifications de la machine virtuelle Java définissent :

- Les concepts du langage Java
- Le format des fichiers .class
- Les fonctionnalités de la JVM
- Le chargement des fichiers .class
- Le byte code
- La gestion des threads et des accès concurrents
- ...

Les fonctionnalités de la JVM décrites dans les spécifications sont abstraites : elles décrivent les fonctionnalités requises mais ne fournissent aucune implémentation ou algorithme d'implémentation. L'implémentation est à la charge du fournisseur de la JVM. Il existe de nombreuses implémentations de JVM dont les plus connues sont celles de Sun Microsystems, IBM, BEA, ...

Le respect strict de ces spécifications par une implémentation de la JVM garantit la portabilité et la bonne exécution du byte code.

Ces spécifications sont consultables à l'url : http://java.sun.com/docs/books/jvms/

Ce chapitre contient plusieurs sections :

♦ La mémoire de la JVM
♦ Le cycle de vie d'une classe dans la JVM
♦ Les ClassLoaders

♦ Le bytecode
♦ Le compilateur JIT
♦ Les paramètres de la JVM HotSpot
♦ Les interactions de la machine virtuelle avec des outils externes

44.1. La mémoire de la JVM

Pour faciliter la gestion de la mémoire, Java propose de simplifier la vie des développeurs :

- Il n'est pas possible d'allouer de la mémoire explicitement : c'est la création d'un nouvel objet avec l'opérateur new qui alloue la mémoire requise
- La JVM dispose d'un ramasse miettes qui se charge de libérer la mémoire des objets inutilisés

La machine virtuelle Java utilise un processus de récupération automatique de la mémoire des objets inutilisés nommé ramasse miettes (Garbage Collector en anglais). Les objets inutilisés sont les objets dont aucun autre objet ne possède de référence sur lui.

Ceci permet d'éviter aux développeurs d'avoir à le faire explicitement dans le code mais possède au moins deux inconvénients :

- il n'est pas possible de connaître le moment ou la mémoire d'un objet sera libérée
- le ramasse miettes ne dispense pas le développeur de connaître son mode de fonctionnement et de prendre quelques précautions pour éviter les fuites de mémoires

Le ramasse miettes est une fonctionnalité de la machine virtuelle qui peut mettre en oeuvre plusieurs algorithmes pour rechercher les objets inutilisés et récupérer automatiquement la mémoire de ces objets. Chaque JVM implémente son propre ramasse miettes en utilisant un ou plusieurs algorithmes.

Une JVM 32bits utilise un adressage sur 32 bits ce qui lui permet de gérer jusqu'à 4 Go de mémoire.

44.1.1. Le Java Memory Model

Les règles de gestion de la mémoire dans une JVM sont définies dans le JMM (Java Memory Model). Initialement ces règles sont définies dans la Java Specification Langage : elles ont été revues dans la JSR 133

44.1.2. Les différentes zones de la mémoire

Le stockage des données dans la JVM est opéré dans différentes zones réparties en deux grandes catégories :

- Les zones de mémoire dont la durée de vie est égale à celle de la JVM : elles sont créées au lancement de la JVM et sont détruites à son arrêt
- Les zones de mémoire liées à un thread dont la durée de vie est égale à celle du thread concerné

Plusieurs zones de mémoire sont utilisées par la JVM :

- les registres (register) : ces zones de mémoires sont utilisées par la JVM exclusivement lors de l'exécution des instructions du byte code.
- une ou plusieurs piles (stack)
- un tas (heap)
- une zone de méthodes (method area)

44.1.2.1. La Pile (Stack)

Chaque thread possède sa propre pile qui contient les variables qui ne sont accessibles que par le thread telles que les variables locales, les paramètres, les valeurs de retour de chaque méthode invoquée par le thread.

Seules des données de type primitif et des références à des objets peuvent être stockées dans la pile. La pile ne peut pas contenir d'objets.

La taille d'une pile peut être précisée à la machine virtuelle.

Si la taille d'une pile est trop petite pour les besoins des traitements d'un thread alors une exception de type StackOverflowError est levée.

Si la mémoire de la JVM ne permet pas l'allocation de la pile d'un nouveau thread alors une exception de type OutOfMemoryError est levée.

44.1.2.2. Le tas (Heap)

Cette zone de mémoire est partagée par tous les threads de la JVM : elle stocke toutes les instances des objets créés.

Tous les objets créés sont obligatoirement stockés dans le tas (heap) et sont donc partagés par tous les threads. Comme les tableaux sont des objets en Java, les tableaux sont stockés dans le tas même si ce sont des tableaux de types primitifs.

La libération de cet espace mémoire est effectuée grâce à un mécanisme automatique implémenté dans la JVM : le ramasse miettes (garbage collector). Le ou les algorithmes utilisés pour l'implémentation du ramasse miettes sont à la discrétion du fournisseur de la JVM.

La taille du tas peut être fixe ou variable durant l'exécution de la JVM : dans ce dernier cas, une taille initiale est fournie et cette taille peut grossir jusqu'à un maximum défini.

Si la taille du heap ne permet pas le stockage d'un objet en cours de création, alors une exception de type OutOfMemoryException est levée.

44.1.2.3. La zone de mémoire "Method area"

Cette zone de la mémoire, partagée par tous les threads, stocke la définition des classes et interfaces, le code des constructeurs et des méthodes, les constantes, les variables de classe (variables static) ...

Comme pour la pile, seules des données de type primitif ou des références à des objets peuvent être stockées dans cette zone de mémoire. La différence est que cette zone de mémoire est accessible à tous les threads. Il est donc important dans un contexte multi-threads de sécuriser l'accès à une variable static même si elle de type primitif.

44.1.2.4. La zone de mémoire "Code Cache"

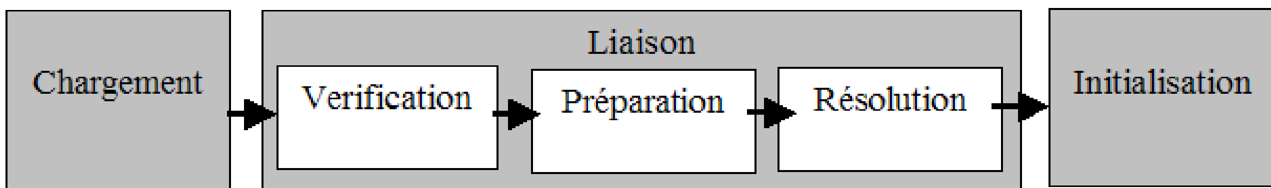
Cette zone de la mémoire stocke le résultat compilé du code des méthodes. La taille par défaut est généralement de 32Mo.

44.2. Le cycle de vie d'une classe dans la JVM

Une classe ou une interface suit un cycle de vie particulier dans la machine virtuelle de son chargement à son retrait.

1. chargement (loading)
2. liaison (linking)

3. initialisation (initialization)
4. instanciation (instantiation)
5. récupération de la mémoire (garbage collection)
6. finalisation (finalization)
7. déchargement (unloading)



Chaque étape est dédiée à une tâche spécifique :

- chargement (load) : permet de lire le byte code dans la machine virtuelle
- liaison (link) : permet de rendre utilisable le byte code. Cette étape est composée de trois processus (vérification, préparation, résolution)
  - La vérification (verify) permet de s'assurer que le byte code est compatible avec la machine virtuelle
  - La préparation (prepare) effectue l'allocation mémoire nécessaire à la classe
  - La résolution (resolve) transforme les références symboliques du constant pool en références mémoire.
- Initialisation (initialize) : initialisation des valeurs des variables.

44.2.1. Le chargement des classes

La machine virtuelle charge, lie et initialise les classes et interfaces requises à l'exécution.

Le démarrage d'une application commence par le chargement de sa classe principale (celle fournie en paramètre de la JVM)

Toutes les classes utilisées pour l'instanciation de cette classe et celles utilisées dans sa méthode main() sont chargées à leur première utilisation.

Un classloader est un objet qui charge dynamiquement et initialise des classes et interfaces Java requises par la JVM lors de l'exécution d'une application. Un classloader hérite de la classe java.lang.ClassLoader.

Un classloader effectue généralement plusieurs opérations pour charger une classe

- Vérification si la classe est déjà chargée et initialisée
- Tentative de chargement du bytecode
- Si le chargement réussi, initialisation du bytecode dans la JVM

Le chargement des classes s'effectue en respectant un modèle de délégation de la responsabilité du chargement. Chaque classloader doit déléguer le chargement de la classe à son classloader père : si ce dernier ne peut mener à bien l'opération alors c'est le classloader lui-même qui tente le chargement.

La méthode loadClass() de la classe ClassLoader exécute par défaut les traitements suivants :

- si la classe est déjà chargée alors elle la renvoie
- sinon délégation du chargement au classloader père
- si la délégation du chargement échoue alors la méthode findClass() est invoquée pour tenter de charger la classe

C'est pour cette raison qu'il n'est pas recommandé lors de la création d'un classloader de redéfinir la méthode loadClass() mais de redéfinir la méthode findClass().

44.2.1.1. La recherche des fichiers .class

La JVM recherche et charge les classes requises dans un ordre bien précis grâce à la délégation des classloaders :

- Les classes de bootstrap (bootstrap classes) qui sont les classes fournies avec la plate-forme Java SE dans le fichier rt.jar
- Les classes d'extension (extension classes) qui sont packagées sous forme de fichiers .jar et stockées dans le répertoire lib/ext du JRE
- Les classes d'utilisateurs (user classes) qui sont écrites par les développeurs ou des tiers

Les classes de bootstrap et d'extension n'ont pas besoin d'être précisées explicitement : elles sont trouvées automatiquement. Les autres classes doivent être précisées en utilisant le classpath.

Les classes des outils contenues dans le fichier tools.jar doivent être ajoutées explicitement dans le classpath pour pouvoir être utilisées.

Les classes de bootstrap sont les classes fournies avec la plate-forme Java. Elles sont principalement dans le fichier rt.jar mais aussi dans quelques fichiers .jar stockés dans le répertoire lib du JRE. L'ensemble des classes de bootstrap est précisé dans la propriété sun.boot.class.path de la JVM.

Même si cela n'est pas recommandé, il est possible de modifier la propriété sun.boot.class.path en utilisant l'option non standard -Xbootclasspath pour définir sa valeur ou ajouter des éléments en début ou en fin de liste.

Le support des classes d'extension a été ajouté dans Java 1.2. Ces bibliothèques permettent d'enrichir les API de base de Java : il faut donc utiliser ce mécanisme de façon judicieuse.

Les classes d'extension sont des extensions de la plate-forme Java qui sont stockées dans le répertoire lib/ext du JRE. Seules les bibliothèques (.jar ou .zip) sont prises en compte. Il n'est pas possible de préciser ou modifier ce chemin. L'ordre de chargement d'une classe contenue dans plusieurs bibliothèques de ce répertoire n'est pas prévisible.

A partir de Java 1.6, il est possible d'utiliser la variable d'environnement java.ext.dirs pour préciser un ou plusieurs répertoires qui permettront le stockage des extensions. Ceci permet d'utiliser ces répertoires par plusieurs JDK sans être obligé de dupliquer les fichiers .jar dans chaque sous répertoire lib/ext de chaque JRE.

Les classes d'utilisateurs sont écrites en reposant sur les classes de bootstrap et d'extension. Pour les trouver, la JVM utilise le classpath qui contient un ensemble de répertoires, et de bibliothèques contenant des classes sous la forme de fichiers .jar et/ou .zip.

Il faut mettre dans le classpath l'entité (répertoire ou bibliothèque) qui contient la classe pleinement qualifiée à utiliser.

Exemple :

- Si une classe com.jmdoudoux.fr.test.MaClasse est stockée dans le répertoire monapp/classes alors le répertoire monapp/classes doit être ajouté au classpath pour utiliser la classe
- Si une classe com.jmdoudoux.fr.test.MaClasse est stockée dans la bibliothèque monapp.jar alors le fichier monapp.jar doit être ajouté au classpath.

Le séparateur des différents éléments du classpath dépend de la plateforme d'exécution (; sous Windows et : sous Unix).

Le classpath peut être obtenu grâce à la variable d'environnement java.class.path de la JVM.

Le classpath peut être précisé de plusieurs façons :

- Par défaut, il ne contient que le répertoire courant « . »
- La variable d'environnement système CLASSPATH
- L'option -cp ou -classpath des outils en ligne de commande
- L'option -jar qui précise une bibliothèque : dans ce cas c'est cette dernière qui doit préciser le classpath

44.2.1.2. Le chargement du byte code

La JVM demande au classloader de rechercher et charger le byte code d'une classe uniquement à sa première utilisation.

Le processus de chargement est composé de trois étapes :

- ouverture d'un flux pour la lecture du byte code
- analyse du byte code et création de données dans la zone de méthode
- création d'une instance de la classe java.lang.class pour la classe

La source du flux n'est pas imposée et peut être un fichier .class local, un fichier .class sur le réseau, une archive (jar ou zip), une génération à la volée, ...

L'instance de la classe Class créée permet une interaction entre une application et la représentation interne de la classe : elle permet par exemple d'obtenir des informations sur la classe.

44.2.2. La liaison de la classe

La liaison de la classe comporte trois étapes :

- La vérification
- La préparation
- La résolution

La vérification est la première étape du processus de liaison : elle permet de s'assurer que la classe chargée est conforme aux spécifications et qu'elle ne risque pas de dégrader la machine virtuelle. La vérification consiste donc en une analyse de la structure et des informations de la classe. Par exemple, pour un fichier .class : vérifier qu'il commence par le nombre magique CAFEBABE, la longueur du fichier, la structure des données, ...

Les spécifications de la JVM détaillent une liste d'exceptions et d'erreurs qui doivent être levées lors de cette étape.

La vérification effectue de nombreux contrôles sur le byte code tel que :

- vérifie les instructions (utilisation d'instructions valides, véracité des sauts, ...)
- vérifie les déclarations d'entités du constant pool (numéro de classes, de méthodes, de champs, ...)
- recherche les classes mères et vérifie que toutes les classes héritent de la classe Object (sauf la classe Object elle-même).
- vérifie que les classes final n'ont pas de classes fille
- vérifie que les méthodes final ne sont pas réécrites
- vérifie que les méthodes des interfaces implémentées soient définies
- vérifie que deux méthodes n'ont pas la même signature
- ...

Certains de ces contrôles nécessitent des informations sur les classes parentes ou sur d'autres classes utilisées qui seront alors chargées mais pas initialisées.

Tous ces contrôles peuvent paraître redondants avec ceux effectués par le compilateur lors de la génération du byte code mais en fait, il est tout à fait possible que le byte code ait été altéré, généré à la volée ou que le compilateur possède un ou plusieurs bugs.

Un mécanisme, déjà utilisé depuis longtemps par Java ME, permet d'ajouter des informations de prévérification lors de la compilation. Ainsi l'étape de validation du byte code est plus rapide à s'exécuter. Depuis la version 6 de Java, le compilateur Java inclut une étape de prévérification qui ajoute des informations dans le fichier.class (StackMap et StackMapTable).

Durant l'étape de préparation, la machine virtuelle alloue la mémoire requise par chaque champs et initialise leurs valeurs avec la valeur par défaut de leur type respectif.

int	0

long	0l
short	0
char	'\u0000'
byte	(byte) 0
float	0.0f
double	0.0d
object	null
boolean (int)	false (0)

Cette étape n'exécute aucun code Java : les valeurs de chaque champ ne sont déterminées que lors de la phase d'initialisation.

Remarque : la machine virtuelle ne définit pas le type booléen. Elle utilise le type int pour sa représentation interne et initialise donc sa valeur à 0 qui correspond à false.

L'étape de résolution permet de rechercher les classes, les interfaces et les membres possédant une référence symbolique dans le constant pool. La résolution permet de remplacer ces références symboliques par des références concrètes.

44.2.3. L'initialisation de la classe

Ce processus a pour rôle d'initialiser les variables de classe avec leur valeur initiale tel que définit dans le code source. La valeur initiale peut être définie de deux façons :

- lors de la déclaration du champ static
- dans un bloc d'initialisation static

Exemple :

```
public class MaClasse {
  static List maListe1 = new ArrayList() ;
  static List maListe2 = null;

  static {
    maListe2 = new ArrayList();
  }
}
```

Ces traitements d'initialisation sont regroupés par le compilateur dans une méthode nommée <clinit (class initialization method). Ces traitements ne concernent que l'exécution de code Java : l'initialisation à l'aide de constantes n'est pas reprise dans cette méthode.

Cette méthode ne peut être invoquée que par la machine virtuelle. Les traitements d'initialisation contenus dans la méthode sont dans l'ordre utilisé dans le code source.

L'initialisation d'une classe implique au préalable l'initialisation de sa classe mère si cela n'a pas déjà été fait et ainsi de suite jusqu'à la classe Object : ainsi toutes les classes mères sont initialisées avant la classe elle même.

Une classe n'a pas obligatoirement de méthode <clinit() : si la classe ne contient aucune variable de classe ou que toutes ces variables sont déclarées finales avec une valeur constante, elle ne possédera pas de méthode <clinit()

Les spécifications de la JVM imposent que l'initialisation d'une classe intervienne à sa première utilisation active :

- création d'une nouvelle instance en utilisant l'opérateur new
- création d'un tableau de la classe

* utilisation d'un membre de la classe qui ne soit pas hérité ni ne soit une constante
* utilisation d'une de ses sous classes (l'initialisation d'une classe impose l'initialisation de toutes ses super classes).

44.2.4. Le chargement des classes et la police de sécurité

L'utilisation d'un classloader implique la mise en oeuvre de la police de sécurité qui lui est associée.

Le simple fait d'utiliser une classe provoque son chargement à sa première utilisation mais il est possible de demander explicitement le chargement d'une classe en invoquant la méthode loadClass() du classloader d'un objet.

Sans police de sécurité, toutes les classes sont considérées comme sûres par défaut.

Même avec une police de sécurité, les classes du bootstrap sont toujours considérées comme sûres.

La police de sécurité repose sur la configuration de la police de sécurité globale et celle de l'application. Par défaut dans la police de sécurité globale, les classes d'extension sont toujours sûres et les autres classes possèdent quelques restrictions.

44.3. Les ClassLoaders

Le contrôle sur le chargement d'une classe permet notamment de mettre en oeuvre certaines techniques avancées telles que la modification du byte code, son instrumentation, son cryptage, ...

Les classloaders étant responsables du chargement d'une classe et comme un ClassLoader est une classe, il existe un classloader particulier, le classloader de bootstrap, qui est implémenté en code natif et qui charge les classes de base de Java dont la classe ClassLoader.

Un autre classloader est dédié au chargement des classes d'extensions (celles des bibliothèques stockées dans le sous répertoire lib/ext du JRE ou à partir de Java 6 celles définies par la propriété java.ext.dirs qui par défaut pointe sur le sous répertoire lib/ext du JRE).

Le troisième classloader créé automatiquement est celui qui permet de charger les autres classes en particulier celles définies dans le classpath : il se nomme classloader d'application. Il permet le chargement des classes définies dans la propriété java.class.path qui par défaut correspond à la variable d'environnement système CLASSPATH.

Les classloaders ont une organisation hiérarchique permettant la mise en oeuvre d'un mécanisme de délégation du chargement d'une classe : un classloader demande toujours à son classloader père d'essayer de charger la classe.

Le mécanisme de délégation permet de s'assurer qu'une classe sera chargée par le classloader qui lui est dédié :

* Les classes de bootstrap sont toujours chargées par le classloader de bootstrap
* Si la classe n'est pas chargée par le classloader de bootstrap, alors le classloader d'extension tente de charger la classe
* Si la classe n'est pas chargée par le classloader d'extension alors le classloader d'application tente de charger la classe
* Si la classe n'est pas chargée par le classloader d'application et qu'aucun classloader dédié n'est définit alors une exception de type ClassNotFoundException est levée

Une classe est associée au classloader qui l'a chargée. Une fois une classe chargée, celle-ci est identifiée par son nom et son classloader. Ainsi, deux classes de même nom chargées par deux classloaders différents sont considérées comme différentes par la JVM.

Si une classe C1 utilise une classe C2 qui n'est pas encore chargée, alors le classloader par défaut pour charger C2 sera celui de C1. Ainsi une classe de bootstrap ne peut pas utiliser une classe du classpath sauf si c'est le classloader system ou un classloader dédié qui est utilisé.

Un thread est associé à un classloader : pour obtenir une référence sur ce classloader, il faut utiliser la méthode getContextClassLoader(). C'est en général le classloader de la classe qui a démarré le thread. Il est parfois nécessaire d'utiliser le classloader du thread notamment avec les servlets qui sont généralement chargées par un classloader dédié du conteneur web. Ce classloader ne permet généralement que de charger des classes contenues dans l'application web, ce qui lui interdit le chargement des classes du classpath.

44.3.1. Le mode de fonctionnement d'un ClassLoader

Dans une JVM, il existe deux classloaders par défaut :

- Le classloader de bootstrap utilisé uniquement par la JVM
- Le classloader système utilisé pour charger les autres classes

Il est aussi possible de définir son propre classloader.

Le classloader possède une méthode loadClass() qui permet de charger une classe à partir de son nom de binaire. Le nom de binaire de la classe correspond au nom pleinement qualifié de la classe incluant le signe $ et l'incrémentation pour les classes anonymes.

Exemple :

java.lang.String
com.jmdoudoux.test.monapp.MonApp
com.jmdoudoux.test.monapp.MonApp$1

La méthode loadClass() lève une exception de type ClassNotFoundException si la classe n'est pas trouvée.

Il y a plusieurs façons pour forcer le chargement d'une classe par un classloader :

- Utiliser explicitement la méthode loadClass() du classloader qui peut lever une exception de type ClassNotFoundException

Exemple :

```
package com.jmdoudoux.test.classloader;

public class TestClassLoader1a {

  public static void main(
      String[] args) {
    try {
      System.out.println(
          TestClassLoader1.class.getClassLoader().loadClass("java.lang.Number"));
    } catch (ClassNotFoundException e) {
      e.printStackTrace();
    }

  }

}
```

- Utiliser la méthode statique Class.forName() qui peut lever une exception de type ClassNotFoundException

Exemple :

```
package com.jmdoudoux.test.classloader;

public class TestClassLoader1b {

  public static void main(
      String[] args) {
    try {
```

```
      System.out.println(Class.forName("java.lang.Number"));
    } catch (ClassNotFoundException e) {
      e.printStackTrace();
    }
  }

}
```

- Utiliser la notation .class qui peut lever une exception de type ClassNotFoundError

Exemple :

```
package com.jmdoudoux.test.classloader;

public class TestClassLoader1c {

  public static void main(
      String[] args) {
    System.out.println(Number.class);
  }

}
```

Chaque objet chargé par un classloader conserve une référence sur ce dernier : la méthode getClassLoader() de la classe Class permet d'obtenir cette référence.

Seule la JVM peut utiliser le classloader de bootstrap : ainsi l'appel de la méthode getClassLoader() d'une classe chargée par le classloader de bootstrap renvoie null.

La classe ClassLoader propose la méthode statique getSystemClassLoader() pour obtenir le classloader système. Sauf création d'un classloader dédié, c'est ce classloader qui charge les classes utilisateurs.

Exemple :

```
package com.jmdoudoux.test.classloader;

public class TestClassLoader4 {

  public static void main(
      String[] args) {
    System.out.println(String.class.getClassLoader());
    System.out.println(ClassLoader.getSystemClassLoader());
    System.out.println(TestClassLoader4.class.getClassLoader());
  }

}
```

Résultat :

```
null
sun.misc.Launcher$AppClassLoader@11b86e7
sun.misc.Launcher$AppClassLoader@11b86e7
```

La classe URLClassLoader est un classloader qui charge des classes à partir d'une ou plusieurs URL fournies en paramètre. Ces urls peuvent correspondre à des répertoires ou à des fichiers jar.

Exemple : le fichier c:\java\test.jar contient la classe com.jmdoudoux.test.MaClasse

```
package com.jmdoudoux.test.classloader;

import java.net.MalformedURLException;
import java.net.URL;
import java.net.URLClassLoader;
```

```
public class TestClassLoader5 {

  public static void main(
      String[] args) {
    try {
      URLClassLoader loader = new URLClassLoader(new URL[] {
        new URL("file:///C:/java/test.jar") });
      Class<?> maClasseClass = loader.loadClass("com.jmdoudoux.test.MaClasse");
      System.out.println(maClasseClass);
    } catch (MalformedURLException e) {
      e.printStackTrace();
    } catch (ClassNotFoundException e) {
      e.printStackTrace();
    }
  }

}
```

Les classloaders assurent qu'une même classe n'est chargée qu'une seule fois par une même hiérarchie de classloaders.

Exemple :

```
package com.jmdoudoux.test.classloader;

public class TestClassLoader3 {

  public static void main(
      String[] args) {

    try {
      boolean resultat = (String.class == Class.forName("java.lang.String"));
      System.out.println("comparaison = "+resultat);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
comparaison = true
```

Le classloader permet aussi de charger des ressources grâce à plusieurs méthodes :

Méthode	Rôle
URL getResource(String name)	Renvoie l'url d'une ressource trouvée par le classloader
InputStream getResourceAsStream(String name)	Renvoie un flux pour lire la ressource
URL getSystemResource(String name)	Méthode statique utilisant le classloader système qui renvoie l'url d'une ressource trouvée
InputStream getSystemResourceAsStream(String name)	Méthode statique utilisant le classloader système qui renvoie un flux pour lire la ressource

Le chargement des ressources en utilisant le classloader est obligatoire par exemple pour charger une ressource incluse dans un fichier .jar.

L'option -verbose:class de la JVM permet de demander l'affichage d'informations sur le chargement des classes.

Résultat :

```
...
[Loaded java.lang.StrictMath from shared objects file]
```

```
[Loaded sun.security.provider.NativePRNG from shared objects file]
[Loaded sun.misc.CharacterDecoder from shared objects file]
[Loaded sun.misc.BASE64Decoder from shared objects file]
[Loaded sun.security.util.SignatureFileVerifier from shared objects file]
[Loaded com.jmdoudoux.test.MaClasse from file:/C:/java/test.jar]
...
```

Cette option peut permettre de déterminer à partir de quelle source une classe est chargée.

44.3.2. La délégation du chargement d'une classe

Il existe une hiérarchie dans les classloaders ce qui permet à un classloader de déléguer le chargement d'une classe à son classloader père. La méthode getParent() de la classe ClassLoader permet de connaître le classloader père. Le classloader de bootstrap ne possède pas de père.

Cette délégation permet notamment de s'assurer que les classes de bootstrap sont chargées par la classeloader de bootstrap. La délégation est effectuée dans la méthode loadClass() qui devrait toujours demander au classloader père de charger la classe.

Exemple :

```
package com.jmdoudoux.test.classloader;

import java.net.MalformedURLException;
import java.net.URL;
import java.net.URLClassLoader;

public class TestClassLoader6 {

  public static void main(
      String[] args) {

    URLClassLoader loader;
    try {
      loader = new URLClassLoader(new URL[] {
        new URL("file:///C:/Program Files/Java/jre1.6.0_03/lib/rt.jar") });
      Class<?> stringClass = loader.loadClass("java.lang.String");
      System.out.println(stringClass.getClassLoader());
      System.out.println(String.class.getClassLoader());
    } catch (MalformedURLException e) {
      e.printStackTrace();
    } catch (ClassNotFoundException e) {
      e.printStackTrace();
    }
  }

}
```

Résultat :

```
null
null
true
```

Bien que le chargement de la classe String soit demandé par une instance de la classe URLClassLoader, la classe est chargée par le classloader de bootstrap. Comme les deux demandes de chargement sont réalisées par le même classloader, les deux classes sont identiques.

Une classe est liée à son classloader : une même classe chargée par deux classloaders sera chargée deux fois (il y aura deux instances de la classe Class correspondante)

Exemple :

```
package com.jmdoudoux.test.classloader;

import java.net.MalformedURLException;
import java.net.URL;
import java.net.URLClassLoader;

public class TestClassLoader7 {

  public static void main(
      String[] args) {
    try {
      URLClassLoader loader1 = new URLClassLoader(new URL[] {
        new URL("file:///C:/java/test.jar") });
      Class<?> maClasseClass1 = loader1.loadClass("com.jmdoudoux.test.MaClasse");
      System.out.println(maClasseClass1);
      URLClassLoader loader2 = new URLClassLoader(
        new URL[] { new URL("file:///C:/java/test.jar") });
      Class<?> maClasseClass2 = loader2.loadClass("com.jmdoudoux.test.MaClasse");
      System.out.println(maClasseClass2);
      System.out.println(maClasseClass1 == maClasseClass2);
    } catch (MalformedURLException e) {
      e.printStackTrace();
    } catch (ClassNotFoundException e) {
      e.printStackTrace();
    }
  }

}
```

Résultat :

```
class com.jmdoudoux.test.MaClasse
class com.jmdoudoux.test.MaClasse
false
```

44.3.2.1. L'écriture d'un classloader personnalisé

L'utilisation d'un classloader dédié peut avoir plusieurs utilités par exemple :

- Personnaliser le chargement du bytecode (par exemple : en l'encryptant après la compilation et en le décryptant au chargement)
- Permettre le rechargement des classes
- Permettre une séparation des classes de plusieurs applications dans une même JVM (exemple avec le conteneur web)
- Enrichir le bytecode pour ajouter des fonctionnalités
- Générer du bytecode à la volée
- ...

Un exemple d'utilisation de classloader personnalisé est dans les conteneurs ou les serveurs d'applications. Généralement, chaque application déployée possède son propre classloader ce qui permet une meilleure isolation des applications exécutées dans la JVM.

Ceci est vrai parce qu'une classe chargée dans la JVM est identifiée par son nom et son classloader. Ceci permet par exemple à un singleton utilisé par plusieurs applications d'être unique par application et non unique dans la JVM puisque chaque application possède son propre classloader.

Ceci permet aussi un rechargement des classes d'une application déployée sans être obligé de relancer la JVM.

L'écriture d'un classloader personnalisé peut permettre de modifier le bytecode : une fois le bytecode chargé le classloader peut le modifier avant de demander son initialisation par la JVM.

Un classloader doit hériter de la classe java.lang.ClassLoader.

La classe ClassLoader possède plusieurs méthodes :

Méthode	Rôle
loadClass()	charger une classe en demandant au préalable au classloader père de réaliser l'opération
findClass()	charger une classe
defineClass()	Ajouter le bytecode de la classe dans la machine virtuelle

Lors de la création de son propre classloader, il faut redéfinir la méthode findClass() plutôt que la méthode loadClass() pour respecter le mécanisme de délégation de chargement de classes des classloaders.

La méthode findClass() ne doit donc être invoquée que si la classe n'a pas pu être chargée par un des classloaders père.

Les données binaires issues de la lecture du fichier et éventuellement enrichies sont passées en paramètres de la méthode defineClass().

Pour utiliser un classloader personnalisé, il faut explicitement demander son utilisation :

- soit en passant en paramètre de la classe méthode forName() de la classe Class une instance du classloader à utiliser
- soit en utilisant la méthode loadClass() du classloader

Comme par défaut le mécanisme de délégation du chargement d'une classe demande au classloader parent de tenter de charger la classe, il faut être sûr que la classe à charger ne se trouve pas dans un classpath particulier (bootstrap classpath, extension classpath, system classpath). Sinon la classe ne sera pas chargée par le classloader personnalisé mais par un de ses classloaders père.

Cette fonctionnalité est utilisée par les conteneurs web qui encourage l'utilisation du sous répertoire WEB-INF/classes plutôt que de mettre les bibliothèques dans le classpath.

44.4. Le bytecode

Le bytecode est un langage intermédiaire entre le code source et le code machine qui permet de rendre l'exécution d'applications Java multiplateforme puisque le byte code est un langage intermédiaire indépendant de tout système d'exploitation.

La JVM fournit un environnement d'exécution pour le byte code en le convertissant en code machine du système d'exploitation utilisé.

Le bytecode peut être modifié avant son exécution par un classloader dédié. Cette modification ou génération de bytecode est par exemple utilisée par :

- Hibernate utilise la génération de bytecode à l'exécution pour générer les classes de persistance
- Certaines implémentations d'AOP tissent leurs aspects en enrichissant le bytecode.

La génération directe de bytecode est plus efficace que la génération de code source puisqu'elle évite l'étape de compilation mais elle est aussi de fait plus compliquée.

Le byte code est défini dans les spécifications de la machine virtuelle Java.

Le byte code est composé de mnémoniques qui réalisent des opérations sur éventuellement un ou plusieurs opérandes. A chaque mnémonique correspond un opcode.

Le compilateur transforme le code source Java en fichiers .class contenant entre autre le byte code.

Lors de la compilation du code source en byte code, le compilateur effectue de nombreuses vérifications notamment sur la syntaxe du code source pour garantir que le byte code produit est valide et qu'il ne risque pas de nuire à la JVM qui va

l'exécuter.

Lors du chargement d'un fichier .class, le classloader effectue des vérifications sur le contenu du fichier afin de s'assurer qu'il ne soit pas en mesure de mettre à mal l'intégrité de la machine virtuelle :

- les 4 premiers octets doivent contenir le chiffre magique (la valeur hexadécimale est "CAFEBABE") qui identifie le fichier comme étant un fichier .class
- chaque classe déclarée final n'est pas sous classée
- tous les attributs et méthodes contiennent une référence dans le pool de constantes (constants pool)
- ...

D'autres langages peuvent être utilisés avec un compilateur dédié pour créer du byte code par exemple :

- Groovy
- JBasic
- Nice
- JRuby
- Scala

44.4.1. L'outil Jclasslib bytecode viewer

Jclasslib est un outil graphique gratuit qui permet de visualiser le byte code contenu dans un fichier .class.

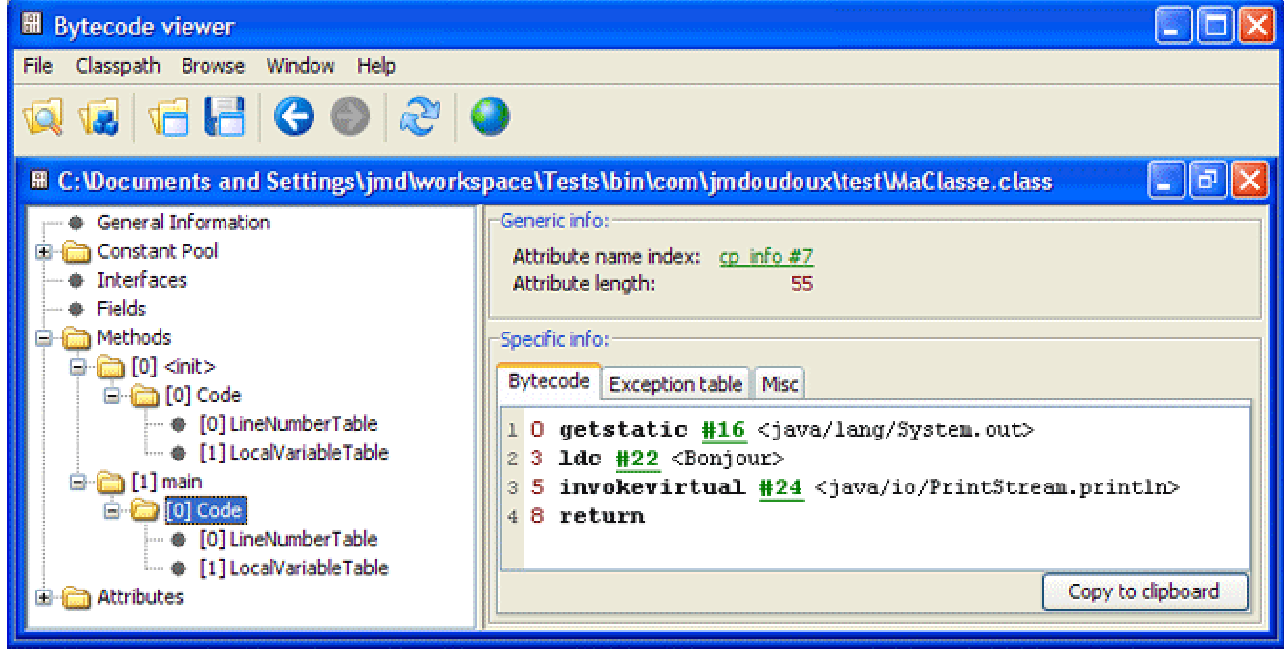
Il peut être téléchargé à l'url http://www.ej-technologies.com/products/jclasslib/overview.html

L'installation se fait sous Windows via un assistant en exécutant le fichier jclasslib_windows_3_0.exe

Il suffit d'exécuter l'outil et d'utiliser l'option « File/Open Class File » en sélectionnant le fichier .class

Exemple :

```
package com.jmdoudoux.test;

public class MaClasse {

  /**
   * @param args
   */
  public static void main(
      String[] args) {
    System.out.println("Bonjour");
  }

}
```



La partie de gauche affiche une vue hiérarchique de la structure du fichier. La partie de droite affiche le contenu de l'élément sélectionné dans la partie de gauche

Jclasslib permet uniquement de visualiser le byte code, il ne permet pas de le modifier.

44.4.2. Le jeu d'instructions de la JVM

La JVM possède un ensemble d'instructions qui sont utilisées pour définir des traitements. Le code source représentant la logique des traitements est compilé par le compilateur pour générer un fichier binaire .class.

Les instructions de la JVM sont des opérations basiques qui combinées permettent de réaliser les traitements.

Une instruction est composée d'un code opération (opcode) suivi d'aucune, une ou plusieurs opérandes qui représentent les paramètres de l'instruction.

Chaque code opération correspond à une valeur stockée sur un octet.

Exemple :

```
package com.jmdoudoux.test;

public class ClasseDeTest {

  public static void main(
      String[] args) {

    for (int i =1; i <=10; i++) {}

  }
}
```

Résultat : utilisation de l'outil de desassemblage javap

```
C:\Documents and Settings\jmd\workspace\Tests\bin\com\jmdoudoux\test>javap -c Cl
asseDeTest
Compiled from "ClasseDeTest.java"
public class com.jmdoudoux.test.ClasseDeTest extends java.lang.Object{
public com.jmdoudoux.test.ClasseDeTest();
  Code:
   0:    aload_0
   1:    invokespecial    #8; //Method java/lang/Object."<init>":()V
```

```
   4:   return

public static void main(java.lang.String[]);
  Code:
   0:   iconst_1
   1:   istore_1
   2:   goto    8
   5:   iinc    1, 1
   8:   iload_1
   9:   bipush  10
   11:  if_icmple       5
   14:  return

}
```

La plupart des instructions sont très basiques. Par exemple, les instructions de l'exemple précédent sont :

iconst_1 : définit une constante entière ayant pour valeur 1
istore_1 : copie la valeur en haut de la pile dans la variable dont l'index est précisé

44.4.3. Le format des fichiers .class

Le format des fichiers .class est décrit dans les spécifications de la JVM.

Un fichier .class est un fichier binaire qui contient :

- un ensemble de structures de données sur la classe elle-même et ses membres (méthodes et champs)
- ...

La suite de cette section sera développée dans une version future de ce document

44.5. Le compilateur JIT

Le byte code est indépendant de toute plate-forme : une fois le code source compilé en byte code, celui-ci peut être exécuté tel quel sur toute plate-forme disposant d'une JVM sous réserve qu'aucun appel à du code natif ne soit fait via l'API JNI.

La JVM se charge alors d'interpréter le byte code lors de son exécution pour le transformer en instruction compréhensible par le processeur de la plate-forme. Ce processus qui assure l'indépendance du byte code vis-à-vis de la plate-forme à aussi l'inconvénient d'être lent car il nécessite une interprétation du byte code et que cette interprétation doit avoir lieu à chaque appel d'une méthode même si cette méthode doit être invoquée plusieurs fois.

L'idée d'un compilateur JIT est de compiler en code natif le byte code d'une méthode, de stocker le résultat de cette compilation et d'exécuter ce code compilé chaque fois que la méthode est invoquée.

Le but d'un compilateur JIT (Just In Time) est donc d'améliorer les performances de l'exécution du byte code.

Ce compilateur est intégré à la JVM pour que son action n'intervienne qu'à l'exécution et préserve la portabilité du byte-code. Le compilateur JIT modifie le rôle de la machine virtuelle qui interprète le byte code en compilant ce dernier à la volée en code natif. Ceci améliore généralement les performances puisqu'une fois le byte code compilé en natif il peut être exécuté directement par le système.

Les méthodes ne sont compilées par le compilateur JIT qu'au moment de leur exécution. Une fois celle-ci compilée, c'est

la version compilée qui sera exécutée au lieu de la version interprétée. L'intérêt du compilateur JIT est donc d'autant plus grand que le nombre de fois où la méthode est invoquée.

Un compilateur JIT est inclus dans la JVM hotspot depuis la version 1.2 de Java.

La performance ajoutée par l'utilisation d'un compilateur JIT est induite par plusieurs faits :

- le code natif s'exécute plus rapidement que le code interprété
- la réutilisation du code déjà compilé nativement est plus rapide que la réinterprétation de chaque ligne de code à chaque invocation de la méthode

Le temps nécessaire au compilateur JIT pour compiler le code peut être pénalisant d'autant que le temps nécessaire au compilateur peut augmenter avec la quantité d'optimisation réalisée par le compilateur.

La machine virtuelle proposée par Sun peut fonctionner selon deux modes. Dans le mode client, c'est la réduction du temps de compilation qui est privilégiée au détriment des optimisations. Dans le mode serveur, c'est l'optimisation qui est privilégiée ce qui allonge le temps de compilation.

44.6. Les paramètres de la JVM HotSpot

La JVM Hotpsot possède des options standards et des options non standards qui peuvent être dépendantes de la plate-forme d'exécution. Les options standards sont décrites dans la section dédiée à la commande java.

Les paramètres non standards sont préfixés par -X : il n'y a aucune garantie sur leur support dans les différentes versions de la JVM. L'option -X permet d'obtenir un résumé des options non standard supportées par la JVM.

Résultat :

```
C:\>java -version
java version "1.6.0_11"
Java(TM) SE Runtime Environment (build 1.6.0_11-b03)
Java HotSpot(TM) Client VM (build 11.0-b16, mixed mode, sharing)

C:\>java -X
    -Xmixed           mixed mode execution (default)
    -Xint             interpreted mode execution only
    -Xbootclasspath:<directories and zip/jar files separated by ;>
                      set search path for bootstrap classes and resources
    -Xbootclasspath/a:<directories and zip/jar files separated by ;>
                      append to end of bootstrap class path
    -Xbootclasspath/p:<directories and zip/jar files separated by ;>
                      prepend in front of bootstrap class path
    -Xnoclassgc       disable class garbage collection
    -Xincgc           enable incremental garbage collection
    -Xloggc:<file>    log GC status to a file with time stamps
    -Xbatch           disable background compilation
    -Xms<size>        set initial Java heap size
    -Xmx<size>        set maximum Java heap size
    -Xss<size>        set java thread stack size
    -Xprof            output cpu profiling data
    -Xfuture          enable strictest checks, anticipating future default
    -Xrs              reduce use of OS signals by Java/VM (see documentation)
    -Xcheck:jni       perform additional checks for JNI functions
    -Xshare:off       do not attempt to use shared class data
    -Xshare:auto      use shared class data if possible (default)
    -Xshare:on        require using shared class data, otherwise fail.

The -X options are non-standard and subject to change without notice.
```

Les principales options non standards sont :

Option	Rôle

-Xint	Désactiver le compilateur JIT : dans ce cas tout le byte code est exécuté en mode interprété uniquement.
-Xbatch	Désactiver la compilation en tâche de fond
-Xbootclasspath:*bootclasspath*	Définir les répertoires, les jar ou les archives zip qui composent les classes de bootstrap. Chaque élément est séparé par un point virgule.
-Xbootclasspath/a:*path*	Ajouter des répertoires, des jar ou des archives zip aux classes de bootstrap
-Xcheck:jni	Effectuer des contrôles poussés sur les paramètres utilisés lors d'appels à des méthodes natives avec JNI. Si un problème est détecté lors de ces contrôles, alors la machine virtuelle est arrêtée avec une erreur fatale. L'activation de cette option dégrade les performances mais renforce la stabilité de la JVM lors des appels à des méthodes natives.
-Xnoclassgc	Désactiver la récupération de la mémoire par le ramasse miettes des classes chargées mais inutilisées. Ceci peut légèrement améliorer les performances mais provoquer un manque de mémoire.
-Xincgc	Active les collectes incrémentales pour le ramasse miettes. Ceci permet de réduire les longs temps de pauses nécessaires au ramasse miettes en réalisant une partie de son activité de façon concomitante avec l'exécution de l'application.
-Xloggc:*file*	Active les traces d'exécution du ramasse miettes dans un fichier de log fourni en paramètre. Il est recommandé d'utiliser un fichier sur le système de fichiers local pour éviter des problèmes de latences réseaux. Cette option est prioritaire sur l'option -verbose:gc si les deux sont fournies à la JVM
-Xms*n*	Permet de préciser la taille initiale du tas. La valeur par défaut dépend du système d'exploitation.
-Xmx*n*	Permet de préciser la taille maximale du tas. La valeur par défaut dépend du système d'exploitation.
-Xprof	Activer l'affichage sur la console de traces de profiling. A défaut de mieux, cette option peut être utilisée dans un environnement de développement mais ne doit pas être utilisée en production car elle dégrade les performances
-Xss*n*	Permet de définir la taille de la pile des threads

Tous les paramètres qui sont préfixés par -XX sont spécifiques à la JVM HotSpot et parfois dépendants de la plate-forme d'exécution.

La syntaxe de ces options dépend de leur type :

options booléennes	la syntaxe pour activer l'option est -XX:+<option> et -XX:-<option> pour la désactiver
options numériques	-XX:<option>=<valeur>. Si la valeur représente une quantité de données, il est possible de préfixer la valeur avec une lettre qui représente l'unité utilisée ('k' ou 'K' pour kilo bytes, 'm' ou 'M' pour mega bytes, et 'g' ou 'G' pour giga bytes)
options littérales	-XX:<option>=<valeur>

La JVM possède selon sa version de nombreuses options -XX dont voici les principales :

Option	Type	Rôle
-XX:DisableExplicitGC	booléen	Empêcher l'invocation explicite de la méthode System.gc()
-XX:ScavengeBeforeFullGC	booléen	Effectuer une récupération de la mémoire de la young generation avant d'effectuer un full garbage collector

-XX:UseConcMarkSweepGC	booléen	Utiliser l'algorithme concurrent mark and sweep pour la récupération de la mémoire de la tenured generation
-XX:UseGCOverheadLimit	booléen	Activer ou non la levée d'une exception de type OutOfMemoryError si la VM passe 98% de son temps dans l'activité du ramasse miettes pour ne récupérer qu'une faible quantité de mémoire. Le but étant d'éviter des traitements longs qui sont quasi inutiles (Depuis Java 6)
-XX:UseParallelGC	booléen	Demander l'utilisation de l'algorithme parallel collector par le ramasse miettes (Depuis Java 1.4.1)
-XX:UseParallelOldGC	booléen	Demander l'utilisation de l'algorithme parallel compacting collector par le ramasse miettes (Depuis Java 5 update 6)
-XX:UseSerialGC	booléen	Utiliser l'algorithme serial pour la récupération de la mémoire de la tenured generation (depuis Java 5.0)
-XX:UseThreadPriorities	booléen	Demander l'utilisation des priorités des threads natifs
-XX:MaxHeapFreeRatio	numérique	Préciser le pourcentage maximum de mémoire libre du tas après une récupération de mémoire afin de provoquer une réduction au besoin de la taille du tas
-XX:MaxNewSize	numérique	Préciser la taille maximale de la young generation (depuis Java 1.4)
-XX:MaxPermSize	numérique	Préciser la taille maximale de la permanent generation. La taille par défaut dépend de la plate-forme
-XX:MinHeapFreeRatio	numérique	Préciser le pourcentage minimum de mémoire libre du tas après une récupération de mémoire afin de provoquer une extension de la taille du tas
-XX:NewRatio	numérique	Préciser le ratio de la taille des deux générations (old et tenured). Les valeurs par défaut dépendent de la plate-forme d'exécution
-XX:NewSize	numérique	Préciser la taille de la young generation
-XX:ReservedCodeCacheSize	numérique	Préciser la taille de la zone de mémoire code cache
-XX:SurvivorRatio	numérique	Préciser le ratio de la taille des espaces eden et des deux survivors de la young generation
-XX:ThreadStackSize	numérique	Préciser la taille en kilo octets de la pile d'un thread. La valeur 0 indique d'utiliser la valeur par défaut
-XX:UseFastAccessorMethods	booléen	Demander l'utilisation de la version optimisée des getters
-XX:StringCache	booléen	Activer la mise en cache des chaînes de caractères.
-XX:CITime	booléen	Afficher des informations sur le temps d'exécution du compilateur JIT (depuis Java 1.4)
-XX:ErrorFile	littéral	Préciser le fichier qui va contenir la log en cas d'erreur fatale. (depuis Java 6)
-XX:HeapDumpPath	littéral	Préciser le chemin ou le nom du fichier qui va contenir le dump du tas (depuis Java 1.4.2, Java 5 update 7)
-XX:HeapDumpOnOutOfMemoryError	booléen	Demander la génération d'un dump au format binaire HPROF dans un fichier du répertoire courant dans le cas ou une exception de type OutOfMemoryError est levée. Le nom de ce fichier est de la forme java_pidxxxx.hprof ou xxxx est le pid de la JVM. (depuis Java 1.4.2 update 12 et Java 5.0 update 7)
-XX:OnError	littéral	Demander l'exécution d'un script ou d'une ou plusieurs commandes séparées par un point virgule lorsqu'une erreur fatale survient. (depuis Java 1.4.2 update 9).

		La séquence %p peut être utilisée pour indiquer le process ID (pid) Exemple : `java -XX:OnError="cat hs_err_pid%p.log|mail jmd@test.fr" MomApp`
-XX:OnOutOfMemoryError	littéral	Demander l'exécution d'un script ou d'une ou plusieurs commandes séparées par un point virgule lorsqu'une exception de type OutOfMemomryError est levée. (depuis Java 1.4.2 update 12)
-XX:PrintClassHistogram	booléen	Afficher un histogramme des instances de classes du tas lors de l'appui sur Ctrl-Break (Depuis Java 1.4.2)
-XX:PrintConcurrentLocks	booléen	Afficher une liste des verrous d'accès concurrents (locks) de chaque thread lors de l'appui sur Ctrl-Break (Depuis Java 6)
-XX:PrintCommandLineFlags	booléen	Afficher les options fournies à la JVM via la ligne de commande (depuis Java 5)
-XX:PrintCompilation	booléen	Activer l'affichage de messages d'information lors de la compilation du byte code d'une méthode.
-XX:PrintGC	booléen	Activer l'affichage de messages d'informations lors de l'exécution du ramasse miettes
-XX:PrintGCDetails	booléen	Activer l'affichage de messages d'informations détaillées lors de l'exécution du ramasse miettes (depuis Java 1.4)
-XX:PrintGCTimeStamps	booléen	Afficher un timestamp à chaque exécution du ramasse miettes (depuis Java 1.4)
-XX:PrintTenuringDistribution	booléen	Afficher une liste de la taille des objets ayant survécus aux dernières exécutions du ramasse miettes dans la young generation (Depuis Java 6 pour le parallel collector)
-XX:TraceClassLoading	booléen	Activer l'affichage de messages lors du chargement des classes
-XX:TraceClassUnloading	booléen	Activer l'affichage de messages lors du déchargement des classes
-XX:ShowMessageBoxOnError	booléen	Afficher une demande à l'utilisateur s'il souhaite lancer le débogueur natif (exemple Visual Studio sous Windows) si la JVM rencontre une erreur fatale.

Toutes les options sont décrites à l'url http://java.sun.com/docs/hotspot/VMOptions.html.

Depuis Java 6, il est possible de modifier dynamiquement certaines de ces options en utilisant le MBean HotSpotDiagnostic exposé par la JVM.

Le contenu de ce chapitre concerne la version 1.6 de la JVM de Sun.

Résultat :

```
C:\Documents and Settings\T30>java -version
java version "1.6.0_02"
Java(TM) SE Runtime Environment (build 1.6.0_02-b06)
Java HotSpot(TM) Client VM (build 1.6.0_02-b06, mixed mode, sharing)
```

L'option -X permet d'obtenir de l'aide sur les paramètres de la JVM concernant des paramètres dont la plupart concerne la gestion de la mémoire.

Résultat :

```
C:\>java -X
    -Xmixed           mixed mode execution (default)
    -Xint             interpreted mode execution only
    -Xbootclasspath:<directories and
zip/jar files separated by ;>
                      set search path for
bootstrap classes and resources
    -Xbootclasspath/a:<directories and
zip/jar files separated by ;>
                      append to end of
bootstrap class path
    -Xbootclasspath/p:<directories and
zip/jar files separated by ;>
                      prepend in front of
bootstrap class path
    -Xnoclassgc       disable class garbage collection
    -Xincgc           enable incremental garbage collection
    -Xloggc:<file>    log GC status to a file with time stamps
    -Xbatch           disable background compilation
    -Xms<size>        set initial Java heap size
    -Xmx<size>        set maximum Java heap size
    -Xss<size>        set java thread stack size
    -Xprof            output cpu profiling data
    -Xfuture          enable strictest checks, anticipating future default
    -Xrs              reduce use of OS signals by Java/VM (see
documentation)
    -Xcheck:jni       perform additional checks for JNI functions
    -Xshare:off       do not attempt to use shared class data
    -Xshare:auto      use shared class data if possible (default)
    -Xshare:on        require using shared class data, otherwise fail.
The -X options are
non-standard and subject to change without notice.
```

L'option -Xms permet de préciser la taille initiale du tas (heap) de la JVM

Résultat :

```
-Xms256m
```

Généralement la valeur par défaut de ce paramètre est insuffisante surtout pour des applications serveur.

L'option -Xmx permet de préciser la taille maximale du tas (heap) de la JVM.

Résultat :

```
-Xmx512m
```

La quantité de mémoire peut être précisée avec plusieurs unités :

- 'k' or 'K' pour kilobytes,
- 'm' or 'M' pour megabytes,
- 'g' or 'G' pour gigabytes

La JVM étend automatiquement la taille du tas de la taille précisée par Xms jusqu'à Xmx lorsque le pourcentage de l'espace libre devient inférieur à la valeur précisé par le paramètre -XX:MinHeapFreeRati. Le paramètre -XX:MaxHeapFreeRatio est équivalent mais réduit la taille du tas si le pourcentage d'espace libre est supérieure à celui fournit.

L'option -verbose:gc permet d'afficher des informations sur chaque récupération de mémoire dont chacune sera sur une ligne distincte.

Résultat :

```
[GC 896K->248K(5056K), 0.0057627 secs]
[GC 1144K->343K(5056K), 0.0034792 secs]
[GC 1239K->504K(5056K), 0.0035857 secs]
```

Les valeurs numériques entre -> correspondent à la valeur de mémoire libre avant et après la récupération de mémoire.

Le nombre de secondes indique le temps utilisé par la récupération de mémoire.

Le paramètre -XX:+PrintGCTimesStamps permet d'ajouter en début de ligne un timestamp pour chaque exécution.

Résultat :

```
0.296: [GC 896K->248K(5056K), 0.0057633 secs]
0.439: [GC 1144K->343K(5056K), 0.0033870 secs]
0.548: [GC 1239K->504K(5056K), 0.0035510 secs]
```

L'option -Xnoclassgc permet de désactiver le déchargement d'une classe lorsque plus aucune instance de cette classe n'est présente dans la mémoire de la JVM. Ceci évite d'avoir à recharger la classe.

44.7. Les interactions de la machine virtuelle avec des outils externes

La machine virtuelle propose des interfaces pour permettre sa connexion avec des outils externes de profiling ou de débogage.

44.7.1. L'API Java Virtual Machine Debug Interface (JVMDI)

La suite de cette section sera développée dans une version future de ce document

44.7.2. L'API Java Virtual Machine Profiler Interface (JVMPI)

L'API Java Virtual Machine Profiler Interface (JVMPI) standardise les interactions entre la JVM et un profiler.

C'est une interface bi directionnelle qui définit

- comment la JVM notifie des événements d'exécution (appel de méthodes, création d'objets, démarrage de threads, ...)
- comment un profiler s'abonne aux événements et obtient des informations

Un agent du profiler est exécuté directement dans la JVM sous une forme native.

Pour exécuter l'agent, la JVM doit être lancée avec le paramètre -XrunProfilerLibrary où ProfilerLibrary est le nom de la bibliothèque native de l'agent.

La page de référence de JVMPI est à l'url : http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/jvmpi.html

JVMPI est deprecated à partir de la version 5.0 de Java et n'est plus disponible à partir de la version 6.0 de Java.

44.7.3. L'API Java Virtual Machine Tools Interface (JVMTI)

Dans la version 5.0 de Java, l'API JVMPI est toujours présente mais elle est deprecated. Dans la version 6.0, JVMPI n'est plus disponible. Elle est remplacée par l'API Java Virtual Machine Tools Interface (JVMTI). Cette API est spécifiée dans la JSR 163 (Java Platform Profiling Architecture)

La page de référence de JVMTI est à l'url : http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/index.html

Cette API est composée d'une partie native (en C/C++) et d'une partie en pure Java. Cette API permet le développement d'outils qui vont interroger l'état de la JVM (outil de profiling, monitoring, débogage, ...). Ces outils sont développés sous la forme d'agents.

L'API Java est contenue dans le package java.lang.instrument.

Les spécifications de la version 1.1 sont consultables à l'url http://java.sun.com/javase/6/docs/platform/jvmti/jvmti.html

Le JDK propose plusieurs exemples de mise en oeuvre de JVMTI dans le sous répertoire demo/jvmti du répertoire d'installation du JDK.

44.7.4. L'architecture Java Platform Debugger Architecture (JPDA)

Java Platform Debugger Architecture (JPDA) est une architecture pour les outils de type débogueur.

Cette architecture repose sur deux API :

- Java Virtual Machine Tools Interface (JVMTI) :
- Java Debug Interface (JDI) : cette API Java doit être implémentée par l'outil de débogage.

Le protocole Java Debug Wire Protocol (JDWP) formalise les échanges entre le débogueur et les traitements en cours de débogage.

Les spécifications de JPDA sont consultables à l'url http://java.sun.com/javase/6/docs/technotes/guides/jpda/index.html

Un exemple de mise en oeuvre de JPDA est proposé dans le sous répertoire /demo/jpda du répertoire d'installation du JDK.

44.7.5. Des outils de profiling

Il existe plusieurs profilers open source notamment :

- Netbeans profiler : http://profiler.netbeans.org/
- Eclipse Test & Performance Tools Platform : http://www.eclipse.org/tptp/
- JBoss Profiler : http://www.jboss.org/jbossprofiler/
- Jprof : http://perfinsp.sourceforge.net/jprof.html

Une liste complète des profilers open source est disponible à l'url :

http://java-source.net/open-source/profilers

Il existe aussi plusieurs solutions commerciales notamment JProfiler ou OptimizeIt.

45. La gestion de la mémoire

Ce chapitre va détailler la gestion de la mémoire dans la plate-forme Java SE.

Celle-ci repose en grande partie sur le ramasse miettes ou garbage collector (les deux désignations sont utilisées dans ce chapitre) dont le mode de fonctionnement et la mise oeuvre sont largement détaillés dans ce chapitre.

Ce chapitre contient quelques informations pour obtenir des informations sur la mémoire, sur les différentes exceptions liées à la mémoire et sur les fuites de mémoire.

Ce chapitre contient plusieurs sections :

- Le ramasse miettes (Garbage Collector ou GC)
- Le fonctionnement du ramasse miettes de la JVM Hotspot
- Le paramétrage du ramasse miettes de la JVM HotSpot
- Le monitoring de l'activité du ramasse miettes
- Les différents types de référence
- L'obtention d'informations sur la mémoire de la JVM
- Les fuites de mémoire (Memory leak)
- Les exceptions liées à un manque de mémoire

45.1. Le ramasse miettes (Garbage Collector ou GC)

Le ramasse miettes est une fonctionnalité de la JVM qui a pour rôle de gérer la mémoire notamment en libérant celle des objets qui ne sont plus utilisés.

La règle principale pour déterminer qu'un objet n'est plus utilisé est de vérifier qu'il n'existe plus aucun autre objet qui possède une référence sur l'objet. Ainsi un objet est considéré comme libérable par le ramasse miettes lorsqu'il n'existe plus aucune référence dans la JVM pointant vers l'objet.

Lorsque le ramasse miettes va libérer la mémoire d'un objet, il a l'obligation d'exécuter un éventuel finalizer définit dans la classe de l'objet. Attention, l'exécution complète de ce finalizer n'est pas garantie : si une exception survient durant son exécution, les traitements sont interrompus et la mémoire de l'objet est libérée sans que le finalizer soit entièrement exécuté.

La mise en oeuvre d'un ramasse miettes possède plusieurs avantages :

- elle améliore la productivité du développeur qui est déchargé de la libération explicite de la mémoire
- elle participe activement à la bonne intégrité de la machine virtuelle : une instruction ne peut jamais utiliser un objet qui n'existe plus en mémoire

Mais elle possède aussi plusieurs inconvénients :

- le ramasse miettes consomme des ressources en terme de CPU et de mémoire
- il peut être à l'origine de la dégradation plus ou moins importante des performances de la machine virtuelle

* le mode de fonctionnement du ramasse miettes n'interdit pas les fuites de mémoires si le développeur ne prend pas certaines précautions. Généralement issues d'erreurs de programmation subtiles, ces fuites sont assez difficiles à corriger.

45.1.1. Le rôle du ramasse miettes

Le garbage collector à plusieurs rôles :

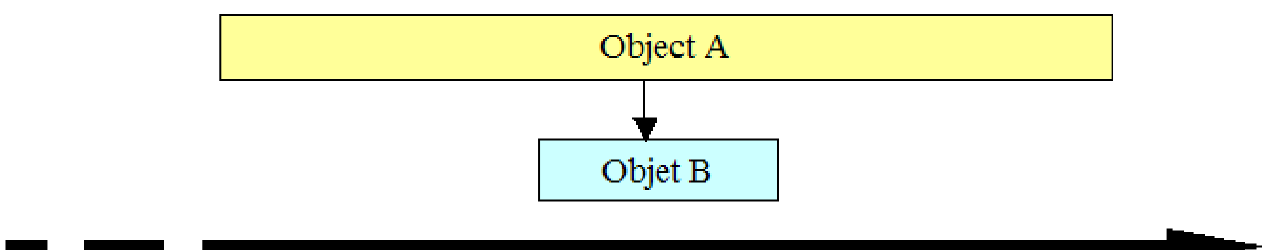
* s'assurer que tout objet dont il existe encore une référence n'est pas supprimé
* récupérer la mémoire des objets inutilisés (dont il n'existe plus aucune référence)
* éventuellement défragmenter (compacter) la mémoire de la JVM selon l'algorithme utilisé
* intervient dans l'allocation de la mémoire pour les nouveaux objets à cause du point précédent

Le ramasse miettes s'exécute dans un ou plusieurs threads de la JVM.

Les objets en cours d'utilisation (dont il existe encore une référence) sont considérés comme "vivant". Les objets inutilisés (ceux dont plus aucun autre objet ne possède une référence) sont considérés comme pouvant être libérés. Les traitements pour identifier ces objets et libérer la mémoire qu'ils occupent se nomment garbage collection. Ces traitements sont effectués par le garbage collector ou ramasse miettes en français.

Le rôle primaire d'un ramasse miettes est de trouver les objets de la mémoire qui ne sont plus utilisés par l'application et de libérer l'espace qu'ils occupent. Le principe général d'exécution du ramasse miettes et de parcourir l'espace mémoire, marquer les objets dont il existe au moins une référence de la part d'un autre objet. Tous les objets qui ne sont pas marqués sont éligibles pour récupérer leur mémoire. Leur espace mémoire sera libéré par le ramasse miettes ce qui augmentera l'espace mémoire libre de la JVM.

Il est important de comprendre comment le ramasse miettes détermine si un objet est encore utilisé ou pas : un objet est considéré comme inutilisé s'il n'existe plus aucune référence sur cet objet dans la mémoire.



Dans l'exemple ci-dessus, un objet A est créé. Au cours de sa vie, un objet B est instancié et l'objet A possède une référence sur l'objet B. Tant que cette référence existe, l'objet B ne sera pas supprimé par le ramasse miettes même si l'objet B n'est plus considéré comme utile d'un point de vue fonctionnelle. Ce cas de figure est fréquent notamment avec des objets des interfaces graphiques et des listeners ou avec des collections.

L'algorithme le plus basic pour un ramasse miettes, parcours tous les objets, marque ceux dont il existe au moins une référence. A la fin de l'opération, tous les objets non marqués peuvent être supprimés de la mémoire. Le gros inconvénient de cet algorithme est que son temps d'exécution est proportionnel au nombre d'objets contenus dans la mémoire. De plus, les traitements de l'application sont arrêtés durant l'exécution du ramasse miettes.

Plusieurs autres algorithmes ont été développés pour améliorer les performances et diminuer les temps de pauses liés à l'exécution du ramasse miettes.

45.1.2. Les différents concepts des algorithmes du ramasse miettes

Plusieurs considérations doivent être prises en compte lors du choix de l'algorithme à utiliser lors d'une collection par le ramasse miettes :

serial ou parallel : avec une collection de type serial, une seule tache ne peut être exécutée à un instant donné même si plusieurs processeurs sont disponibles sur la machine.

Avec une collection de type parallel, les traitements du garbage collector sont exécutés en concomitance par plusieurs processeurs. Le temps global de traitement est ainsi plus court mais ils sont plus complexes et augmentent généralement la fragmentation de la mémoire

stop the world ou concurrent : avec une collection de type stop the world, l'exécution de l'application est totalement suspendue durant les traitements d'une collection. Stop the world utilise un algorithme assez simple puisque durant ces traitements, les objets ne sont pas modifiés durant la collection. Son inconvénient majeur est la mise en pause de l'application durant l'exécution de la collection.

Avec une collection de type concurrent, un ou plusieurs collections peuvent être exécutées simultanément avec l'application. Cependant, une collection de type concurrent ne peut pas réaliser tous ces traitements de façon concurrente et dois parfois réaliser certains de ces traitements sous la forme stop the world.

De plus, l'algorithme d'une collection de type concurrent est beaucoup plus complexe puisque les objets peuvent être modifiés par l'application durant la collection : il nécessite généralement plus de ressources CPU et mémoire pour s'exécuter.

compacting ou non compacting ou copying : une fois la libération de la mémoire effectuée par le garbage collector, il peut être intéressant pour ce dernier d'effectuer un compactage de la mémoire pour mettre de façon contigüe tous les objets alloués et la mémoire libre.

Ce compactage nécessite un certain de traitement mais il accélère ensuite l'allocation de mémoire car il n'est plus utile de déterminer quel espace libre utiliser : s'il y a eu compactage, cette espace correspond obligatoirement à la première zone de mémoire libre rendant l'allocation très rapide.

Si la mémoire n'est pas compactée, le temps nécessaire à la collection est réduit mais il est nécessaire de parcourir la mémoire pour rechercher le premier espace de mémoire qui permettra d'allouer la mémoire requise, ce qui augmente les temps d'allocation de mémoire aux nouveaux objets et la fragmentation de la mémoire.

Il existe aussi une troisième forme qui consiste à copier les objets survivants à différentes collections dans des zones de mémoires différentes (copying). Ainsi la zone de création des objets se vide au fur et à mesure, ce qui rend l'allocation rapide. Le copying nécessite plus de mémoire.

45.1.3. L'utilisation de générations

Suite à diverses observations, plusieurs constats ont été faits sur la durée de vie des objets d'une application en général :

- un nombre particulièrement important d'objets ont une durée de vie courte (exemple : un iterator utilisé dans les traitements d'une méthode)
- la plupart des objets ayant une durée de vie longue ne possède pas de référence vers des objets ayant une durée de vie courte

Il est alors apparu l'idée d'introduire la notion de générations dans le traitement des collections (generational collection). L'idée est de repartir les différents objets dans différentes zones de la mémoire nommées générations selon leur durée de vie. Généralement deux générations principales sont utilisées :

- une pour les jeunes objets (young generation)
- et une pour les objets avec une durée de vie plus longue (old generation ou tenured generation).

L'utilisation de générations possède plusieurs intérêts :

- la portion de mémoire à collecter est réduite
- il est possible d'appliquer des algorithmes différents pour chaque génération
- et d'utiliser un algorithme optimisé selon les caractéristiques d'utilisation d'une génération

Il est facile de conclure que le nombre de collections à réaliser sur la young generation sera beaucoup plus important que sur la old generation. De plus, les collections sur la young generation devraient être rapides puisque la taille devrait être

relativement réduite et que le nombre d'objets sans référence devrait être important. Les collections dans la young generation sont appelés collections mineures (minor collection) puisque très rapide.

Si un objet survit à plusieurs collections, il peut être promu (tenured) dans la old generation. Généralement, la taille de la old generation est plus importante que celle de la young generation. Les collections sur la old generation sont généralement plus longues puisque la taille de la génération est plus importante mais elles sont aussi moins fréquentes.

En effet, une collection dans la old generation n'intervient en général qu'une fois que l'espace mémoire libre dans cette génération devient faible. Une collection dans la old generation étant généralement longue elle est désignée par le terme collection majeure (major collection).

Le but de l'utilisation des générations est de limiter le nombre de collections majeures effectuées.

45.1.4. Les limitations du ramasse miettes

Le ramasse miettes ne résout pas tous les problèmes de mémoires :

- il ne peut empêcher un manque de mémoire si trop d'objets sont à créer dans un espace mémoire trop petit
- il n'empêche potentiellement pas les fuites de mémoire

De plus le ramasse miettes est un processus complexe qui consomme des ressources et nécessite un temps d'exécution non négligeable pouvant être à l'origine de problème de performance.

Une bonne connaissance du mode de fonctionnement du ramasse miettes est obligatoire pour apporter une solution lorsque celui-ci est l'origine de goulets d'étranglement lors de l'exécution de l'application.

Le mécanisme d'allocation de mémoire est aussi lié au garbage collector car il nécessite de trouver un espace mémoire suffisant pour les besoins de l'allocation. Ceci implique pour le garbage collector de compacter la mémoire lors de la récupération de celle-ci pour limiter les effets inévitables de fragmentation de la mémoire.

Le garbage collector est un mécanise complexe mais fiable. Bien que complexe, son fonctionnement doit essayer de limiter l'impact sur les performances de l'application notamment en essayant de limiter son temps de traitement et la fréquence de son exécution. Pour atteindre ces objets, des travaux sont constamment en cours de développement afin de trouver de nouveaux algorithmes. Il peut aussi être nécessaire d'effectuer un tuning du ramasse miettes en utilisant les nombreuses options proposées par la JVM.

La performance du garbage collector est intimement liée à la taille de la mémoire qu'il a géré. Ainsi, si la taille de la mémoire est petite, le temps de traitement du garbage collector sera court mais il interviendra plus fréquemment. Si la taille de la mémoire est grande, la fréquence d'exécution sera moindre mais le temps de traitement sera long. Ainsi le réglage de la taille de la mémoire influe sur les performances du garbage collector et est un des facteurs importants en fonction des besoins de chaque application.

Le ramasse miettes fait son travail dans la JVM mais il se limite aux instances des objets créés par la machine virtuelle. Cependant, dans une application, il peut y avoir d'autres allocations de mémoire qui sont hors du scope des instances d'objets Java.

Ceci concerne des ressources natives du système qui sont allouées par un processus hors du contexte Java. C'est notamment le cas lors de l'utilisation de JNI. Dans ce cas, il faut explicitement demander la libération des ressources en invoquant une méthode dédiée car le ramasse miettes n'a aucun contrôle sur l'espace mémoire de ces entités. Par exemple, certaines classes qui encapsulent des composants de AWT proposent une méthode dispose() qui se charge de libérer les ressources natives du système.

Le traitement du ramasse miettes dans la permanent generation suit des règles particulières :

- les classes chargées par le classloader primordiale ne sont jamais traitées par le ramasse miettes
- les classes chargées par un classloader personnalisé sont considérées comme inutilisées uniquement s'il n'existe plus aucune instance de cette classe et qu'il n'existe plus de référence sur le classloader personnalisé

45.1.5. Les facteurs de performance du ramasse miettes

Pour optimiser les performances du ramasse miettes, il est nécessaire d'avoir des indicateurs sous la forme de métriques :

- throughput : pourcentage de temps dédié par la JVM à l'exécution de l'application
- GC overhead : pourcentage de temps dédié par la JVM à l'exécution du GC
- temps de pause : durée durant laquelle l'exécution de l'application est interrompue à cause de l'exécution des collections du ramasse miettes
- fréquence des collections : nombre de fois ou une collection est exécutée
- footprint : empreinte mémoire du tas
- promptness : durée entre le moment ou l'objet n'est plus utilisé et le moment ou son espace mémoire est libéré

L'importance de ces indicateurs dans le tuning du ramasse miettes dépend du type d'application utilisée, par exemple :

- une application avec une IHM doit limiter les temps de pause
- une application temps réels doit limiter le temps d'exécution et la fréquence d'exécution des collections
- l'empreinte mémoire est primordiale sur un système possédant de faibles ressources
- ...

Le choix de l'algorithme utilisé pour les collections mineures et majeures est important pour les performances globales du ramasse miettes. Il est préférable d'utiliser un algorithme rapide pour la young generation et un algorithme privilégiant l'espace pour la old generation.

Il faut aussi en général privilégier la vitesse d'allocation de mémoire pour les nouveaux objets qui sont des opérations à la demande plutôt que la libération de la mémoire qui n'a pas besoin d'intervenir immédiatement une fois que l'objet n'est plus utilisé sauf si la JVM manque de mémoire.

Pour réaliser des applications pointues ou permettre leur bonne montée en charge, il est important de comprendre les mécanismes utilisés par la JVM pour mettre en oeuvre le ramasse miettes car celui-ci peut être à l'origine de fortes dégradations des performances.

L'algorithme le plus simple d'un ramasse miettes parcours tous les objets pour déterminer ceux dont il n'existe plus aucune référence. Ceux-ci peuvent alors être libérés. Ce temps de traitement du ramasse miettes est alors proportionnel au nombre d'objets présents dans la mémoire de la JVM. Ce nombre peut facilement être très important et ainsi dégrader les performances car durant cette opération l'exécution de tous les threads doit être interrompue.

45.2. Le fonctionnement du ramasse miettes de la JVM Hotspot

Le fonctionnement du ramasse miettes de la JVM Hotspot de Sun Microsystems évolue au fur et à mesure de ces versions et repose sur plusieurs concepts :

- l'utilisation de générations
- plusieurs algorithmes de GC sont proposés dans le but de toujours améliorer les performances du GC selon les besoins

L'idée est toujours de réduire la fréquence d'invocations et les temps de traitements du ramasse miettes.

De nombreux paramètres permettent de configurer le comportement du ramasse miettes de la JVM.

Depuis Java 1.2, le ramasse miettes implémente plusieurs algorithmes et utilise la notion de génération. Les ingénieurs ont constaté que d'une façon générale, il y a deux grandes typologies d'objets créés dans une application :

- les objets à durée de vie courte : exemple des objets créés dans les traitements d'une méthode
- les objets à durée de vie longue

La notion de generation est issue de l'observation du mode de fonctionnement de différentes typologies d'applications relatif à la durée de leurs objets. Ainsi, il a été constaté que de nombreux objets avaient une durée de vie relativement courte.

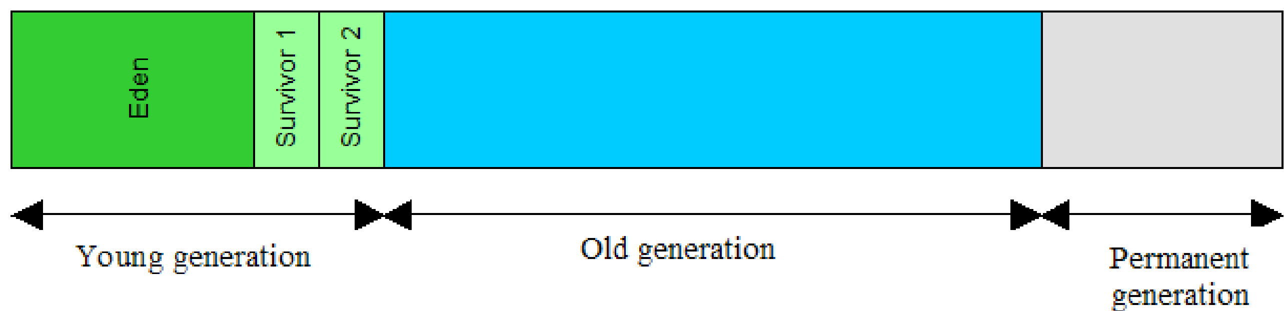
La notion de génération divise la mémoire de la JVM en différentes portions qui vont contenir des objets en fonction de leur âge. Une grande majorité des objets sont créés dans la generation des objets jeunes (young generation) et meurt dans cette génération.

Ainsi le tas est découpé en générations dans lesquels les objets sont passés au fur et à mesure de l'allongement de leur durée de vie :

- young generation : tous les objets sont créés dans cette generation. L'opération de libération de la mémoire dans cette génération est désigné par le terme collection mineure. Comme le nombre objets qui n'ont plus de référence est important, le temps nécessaire à leur libération est très rapide. Après avoir survécu à plusieurs collections mineures, l'objet est promu dans la old generation
- old generation (tenured generation) : tous les objets de cette génération ont été promus de la young generation. L'opération de libération de la mémoire dans cette génération est désignée par le terme collection majeure (full collection). Le temps nécessaire à une collection majeure est beaucoup plus long : c'est pour cette raison que leur fréquence d'exécution est limitée généralement à un besoin absolu comme le manque d'espace dans la tenured generation

La JVM dispose aussi d'une troisième génération nommée permanent generation qui contient des données nécessaires au fonctionnement de la JVM comme par exemple la description de chaque classes et le code de chaque méthodes.

Sauf pour l'algorithme thoughput collector, le découpage de la mémoire de la JVM est généralement sous la forme ci dessous



La young generation est composée de trois parties :

- eden : les objets créés le sont dans cette partie.
- deux parties nommées survivor : ces deux parties sont alternativement remplies à chaque collection mineure

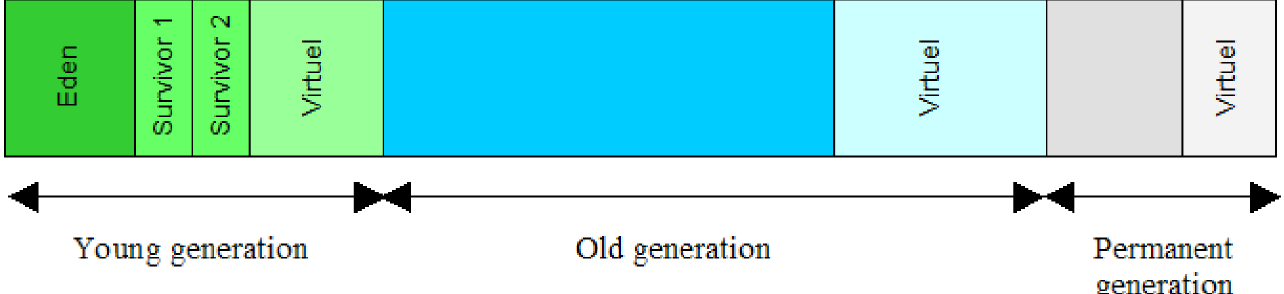
La taille de la young generation doit être suffisante pour permettre à un maximum d'objets d'être libérables entre deux collections mineures.

45.2.1. Les trois générations utilisées

La mémoire d'une JVM Hotspot est composée de trois générations

- young generation : la plupart des objets sont instanciés dans cette génération
- old(tenured) generation : contient les objets qui ont survécu à plusieurs collections de la young generation et qui ont été promus dans cette génération. Cette génération peut aussi contenir des objets de grandes tailles directement créés dans cette génération
- permanent generation : contient des objets nécessaires au fonctionnement de la JVM tels que la description des classes et méthodes et les classes et méthodes elles mêmes

L'organisation des générations est généralement la suivant (sauf pour l'algorithme parallel collector)



Au démarrage de la JVM, tout l'espace mémoire maximum n'est pas physiquement alloué et l'espace de mémoire utilisable en cas de besoin est dit virtuel.

L'espace mémoire utilisé pour stocker les instances d'objets nommé tas (heap) est composé de la young generation et tenured generation

La young generation est composée de plusieurs espaces :

- un espace principal nommé eden : les objets sont créés dans cet espace sauf ceux de grandes tailles
- deux espaces plus petits nommés survivor 1 (to) et 2 (from) : un espace survivor contient les objets qui ont survécus à au moins une collection mineure et qui peuvent donc potentiellement être promus dans la old generation. L'un des deux espaces survivor est toujours vide pendant que l'autre est utilisé par une collection. Il change alternativement leur rôle.

Lorsque la young generation se remplit au point d'être pleine, une collection mineure est effectuée.

Lorsque la young generation est remplie, une collection mineure est exécutée par le ramasse miettes. Une collection mineure peut être optimisée puisqu'elle part du pré requis que la plupart des objets de la young generation vont être récupérés lors d'une collection mineure. Comme la durée d'une collection dépend du nombre d'objets utiles, une collection mineure doit s'exécuter rapidement.

Un objet toujours vivant après plusieurs collections mineures est promu dans la tenured generation. Si la tenured generation est remplie, une collection majeure est exécutée par le ramasse miettes. Lors d'une collection majeure, le ramasse miettes opère sur l'intégralité du tas (young et tenured generation). Généralement une collection majeure est beaucoup plus longue qu'une collection mineure puisqu'elle implique beaucoup plus d'objets à traiter.

L'allocation d'objets est aussi soumise à des problématiques multi-threads puisque plusieurs threads peuvent demander l'allocation d'objets. Pour tenir compte de ces contraintes et ne pas dégrader les performances, la JVM HotSpot réserve à chaque thread une zone de mémoire de l'espace eden nommée Thread Local Allocation Buffer (TLAB) dans laquelle les objets du thread sont créés. Une synchronisation est cependant nécessaire si le TLAB est plein et qu'il faut en allouer un supplémentaire au thread. Des fonctionnalités sont mises en place pour limiter l'espace inutilisé des TLAB.

Lorsque la old generation ou la permanent generation se remplit, une collection majeure est effectuée, impliquant une collection sur les toutes les générations.

Si l'espace libre de la old generation est insuffisant pour stocker les objets promus de la young generation, alors l'algorithme de collection de la old generation est appliqué sur l'intégralité du tas.

La mémoire de la JVM contient une section nommée génération permanente (Perm Gen). La JVM stocke dans cet espace les classes et leurs méthodes.

Du point de vue d'une collection les instances et les classes sont considérées comme des objets puisque ces deux types d'entités ont une représentation similaire dans la JVM. Le stockage dans la permanent generation des classes se justifie par le fait que les classes sont généralement des objets ayant une durée de vie relativement longue. Ainsi pour limiter le travail du ramasse miettes et pour séparer les entités applicatives de celles de la JVM, les classes sont stockées dans cette generation dédiée.

Les principaux objets qui sont stockés dans la permanent generation sont :

- les méthodes des classes ainsi que leur byte code
- les données lues du fichier .class (exemple : constante pool)

- des objets internes utilisés par la JVM
- ...

Les entités stockées dans cet espace ne sont pas vraiment permanent sauf si l'option -noclassgc est utilisée lors du lancement de la JVM.

La taille de l'espace Permanent est indépendante de la taille du tas.

Des applications qui chargent de nombreuses classes ont besoin d'un espace plus important pour l'espace Permanent que celui proposé par défaut. La taille de l'espace Permanent peut être précisée en utilisant l'option -XX:PermSize au lancement de la JVM. La valeur fournie à ce paramètre sera utilisé pour définir la taille de l'espace Permanent au lancement de la JVM. Il est possible de définir la taille maximale de l'espace Permanent en utilisant l'option -xx:MaxPermSize au lancement de la JVM.

Si la taille de l'espace Permanent n'est pas assez important pour les besoins de la JVM, une exception de type OutOfMemoryError est levée avec dans son message une référence au PermGen.

Actuellement, les collections sur la permanent generation sont toujours de type serial. Lorsqu'une collection est effectuée sur la permanent generation, elle est toujours réalisée avant celle de la tenured generation. Un objet de la permanent generation ne change jamais de generation suite à une collection.

45.2.2. Les algorithmes d'implémentation du GC

La version 1.5 de Java SE propose quatre algorithmes d'implémentation pour le ramasse miettes :

- serial collector : c'est l'algorithme utilisé par défaut qui répond à la majorité des besoins standards pour de petites applications. Ces traitements utilisent un seul thread en mode stop the world
- parallel collector ou throughput collector : cet algorithme est le même que le serial collector mais il utilise une version qui parallélise les traitements de la young generation. Il est recommandé pour une application multi-threadée exécutée sur une machine avec beaucoup de mémoire et plusieurs CPU comme des serveurs d'applications.
- parallel compacting collector : cet algorithme parallélise les traitements de la tenured generation (depuis Java 5u6, amélioré en Java 6)
- concurrent mark sweep collector (CMS) ou concurrent low pause collector : une grande partie des traitements sur la tenured generation se fait de façon concurrente avec l'application, ceci réduit les temps de pause de l'application.

Remarque : L'algorithme incremental low pause collector ou train collector n'est plus supporté depuis la version 1.4.2 de Java.

Tous ces algorithmes reposent sur l'utilisation de générations.

45.2.2.1. Le Serial Collector

Avec un algorithme de type Serial collector, les collections de la young et tenured generation sont faites de manière séquentielle, par un seul processeur, à la façon stop the world : ainsi l'exécution de l'application est suspendue durant l'exécution des collections.

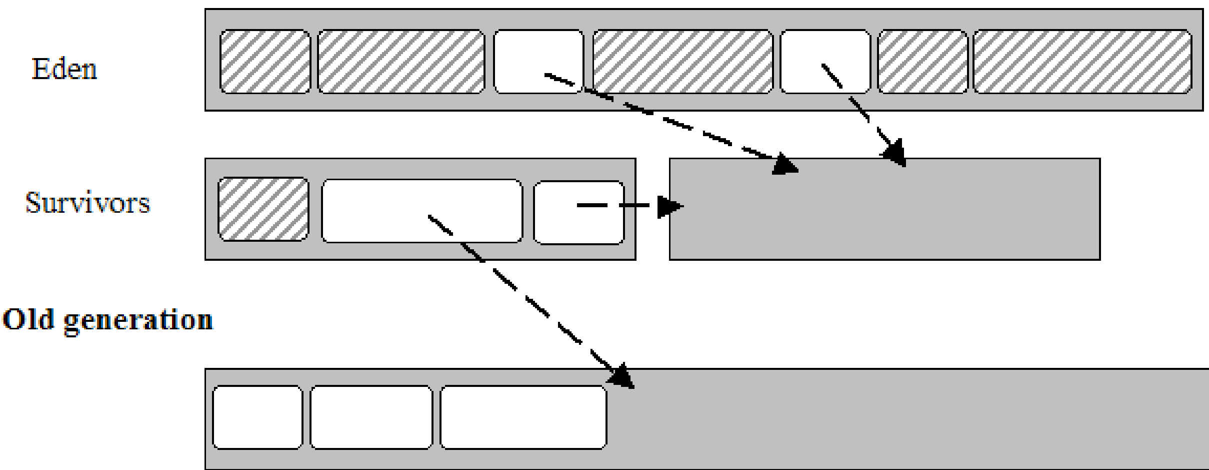
45.2.2.1.1. L'utilisation du serial collector dans la young generation

Les objets utilisés sont déplacés de l'espace eden vers l'espace survivor qui est vide. Les objets qui sont trop gros pour être déplacés dans l'espace survivor sont directement déplacés dans la tenured generation.

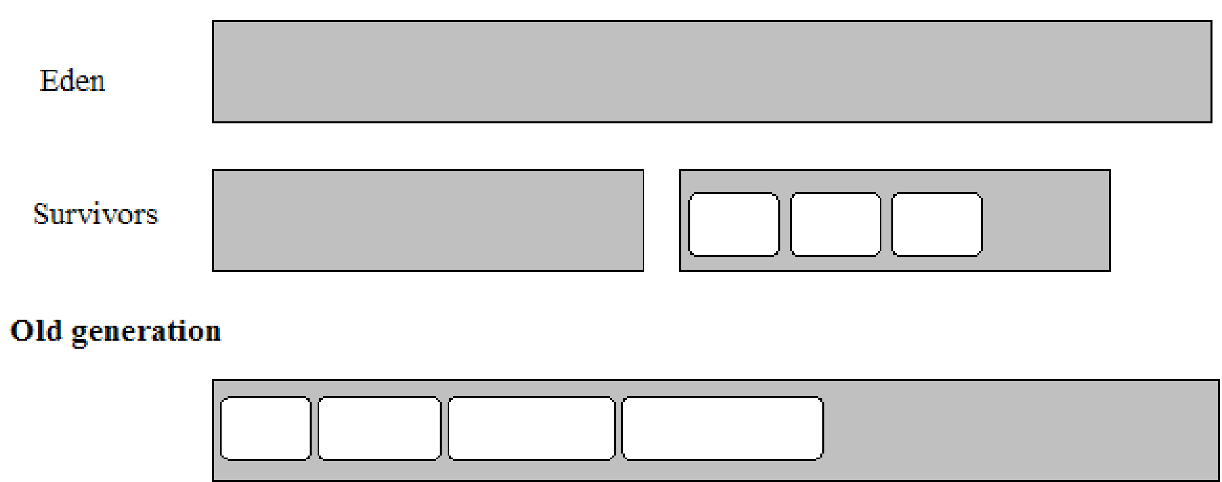
Les objets les plus jeunes de l'espace survivor rempli (survivor from) sont déplacés dans l'autre espace survivor en cours de remplissage (survivor to). Les objets les plus anciens sont déplacés dans la tenured generation. A la fin de la collection l'espace survivor qui était rempli doit être vide. Si l'espace survivor en cours de remplissage ne peut plus recevoir d'objets

alors tous les objets sont directement promus dans la tenured generation sans tenir compte de leur âge.

Après le déplacement des objets en cours d'utilisation, tous les objets qui restent dans l'espace eden et l'espace survivor qui était rempli sont des objets inutilisés dont l'espace peut être récupéré.



A la fin de la collection, l'espace eden et l'espace survivor initialement rempli sont vides. Ainsi dans la young generation, seul l'espace survivor qui a été rempli contient encore des objets. Les deux espaces survivor échangent leur rôle pour la prochaine collection.

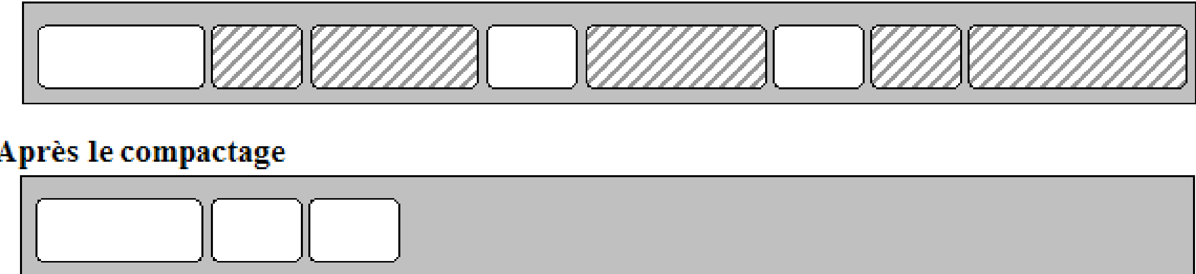


45.2.2.1.2. L'utilisation du serial collector dans la tenured generation

Le serial collector utilise un algorithme de type mark/sweep/compact pour traiter la tenured generation et la permanent generation :

- Durant l'étape mark, le ramasse miettes identifie les objets qui sont encore utilisés
- Durant l'étape sweep, le ramasse miettes identifie les objets qui ne sont plus utilisés, en fait ceux qui n'ont pas été marqués lors de l'étape précédente, et récupère la mémoire qu'ils occupent.
- Durant l'étape compact, le ramasse miettes compacte la mémoire en regroupant tous les objets utilisés au début de la mémoire de la tenured generation. Le compactage de la mémoire permet de rendre la création d'objets dans la tenured generation plus rapide.

Un traitement similaire est effectué dans la permanent generation.



45.2.2.1.3. Le choix de l'utilisation du serial collector

Le serial collector est recommandé pour les applications clientes : ces applications sont généralement peux gourmandes en mémoire et le serial collector peut facilement effectuer un full GC en moins d'une demi seconde.

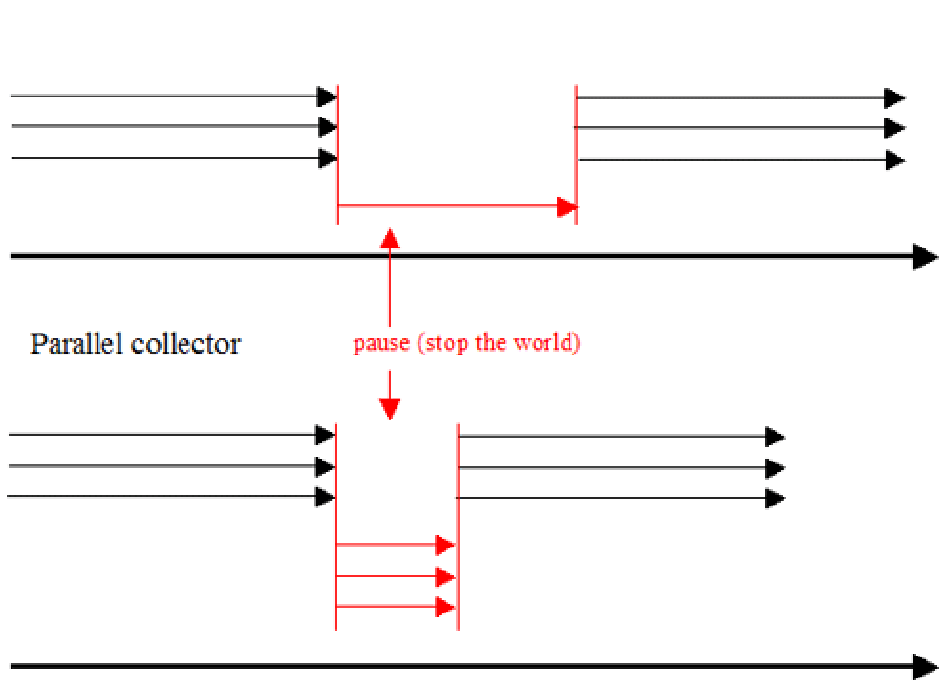
Depuis la version 5 de Java, le serial collector est choisi par défaut sur des machines de type client. Pour demander l'utilisation du serial collector sur des machines de type server, il faut utiliser l'option -XX:+UseSerialGC

45.2.2.2. Le Parallel collector ou throughput collector

Pour profiter des avantages offerts par des machines disposant de plusieurs coeurs ou de plusieurs CPU, le parallel collector, aussi appelé thoughput collector, a été développé. Il permet de réaliser les traitements de la collection en utilisant plusieurs CPU au lieu d'un seul.

45.2.2.2.1. L'utilisation du parallel collector dans la young generation

Le parallel collector utilise une version multi threads de l'algorithme utilisé par le serial collector : c'est toujours un algorithme de type stop the world avec déplacement des objets selon leur ancienneté mais ces traitements sont réalisés en parallèle par plusieurs threads. Le temps de ces traitements est ainsi réduit.



Java 5 propose quelques options supplémentaires pour configurer le comportement souhaité du parallel collector notamment en ce qui concerne le temps maximum de pause de l'application et le throughput.

Le temps maximum de pause de l'application souhaité peut être précisé avec l'option -XX:MaxGCPauseMillis=n ou n représente une valeur en millisecondes. Le parallel collector va tenter de respecter se souhait en procédant à des ajustements de paramètres mais il n'y a aucune garantie sur sa mise en oeuvre. Par défaut, aucun temps maximum de pause n'est défini.

Le throughput souhaité peut être exprimé avec l'option -XX:GCTimeRatio=n ou n entre dans le calcul du ratio entre le temps du ramasse miettes et le temps de l'application selon la formule 1 / (1 + n).

Exemple

-XX:GCTimeRatio=19 correspond à 5% (1 / (1 + 19)) du temps pour le ramasse miettes

La valeur par défaut est 99, ce qui représente 1% du temps pour le ramasse miettes.

Si ce souhait n'est pas atteint, l'algorithme va agrandir la taille des générations pour allonger le délai entre deux collections.

Les priorités dans les souhaits pris en compte par l'algorithme sont dans l'ordre :

- le temps maximum de pause
- le throughput
- l'empreinte mémoire

Pour être pris en compte, les souhaits précédents doivent être atteints avant que l'algorithme ne tente de réaliser le souhait suivant.

Le parallel collector utilise les générations et un algorithme similaire à celui du serial collector pour le traitement de la young generation mais il parallélise ces traitements via plusieurs threads. Par défaut, le parallel collector utilise autant de threads que de processeurs.

Sur une machine avec un seul processeur, le parallel collector est moins performant que le serial collector notamment à cause coût de synchronisation des traitements.

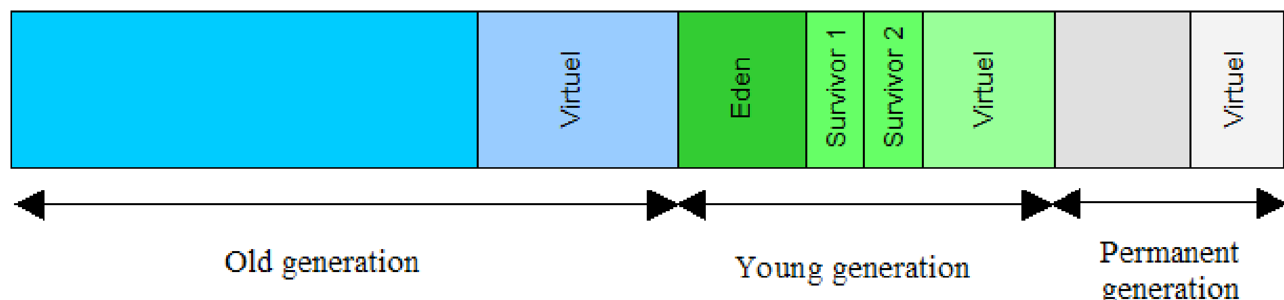
Sur une machine avec deux processeurs, le parallel collector et le serial collector ont des performances similaires.

Le gain en performance sur le temps des collections mineures croit sur des machines avec plus de deux processeurs.

Le nombre de threads utilisés peut être précisé explicitement en utilisant -XX:ParallelGCThreads=n ou n correspond au nombre de threads.

Comme plusieurs threads sont utilisés pour réaliser une collection mineure, il est possible que la tenured generation se fragmente. Chaque thread effectue la promotion d'un objet de la young generation vers la tenured génération dans une portion de cette dernière qui lui est dédiée

Les générations sont organisées de façon particulière dans le tas de la JVM.



A la fin de chaque collection, l'algorithme met à jour ses statistiques et test si les souhaits sont atteints. Si ce n'est pas le cas, l'algorithme va modifier les paramètres du ramasse miettes et la taille du tas et des générations pour tenter d'atteindre

les souhaits exprimés.

Par défaut, la taille d'une generation est augmentée de 20% et réduite de 5%. Ces valeurs peuvent être modifiées en utilisant plusieurs options :

- -XX:YoungGenerationSizeIncrement permet de préciser le pourcentage d'augmentation de la young generation
- -XX:TenuredGenerationSizeIncrement permet de préciser le pourcentage d'augmentation de la tenured generation
- -XX:AdaptiveSizeDecrementScaleFactor permet de préciser un pourcentage de réduction de la taille des générations sous la forme taille d'incrémentation / n

Les demandes explicites d'exécution du ramasse miettes ne rentrent pas le calcul des statistiques.

Si le temps maximum de pause souhaité n'est pas atteint, la taille d'une seule generation est réduite à la fois (celle dont le temps de pause a été le plus long)

Si le throughput souhaité n'est pas atteint, la taille des deux générations est augmentée.

Si la taille minimale et maximale du tas n'est pas précisée au lancement de la JVM alors elles sont déterminées en fonction de la taille de la mémoire physique de la machine. Par défaut, la taille minimale est égale à la taille de la mémoire / 64. Par défaut, la taille maximale est égale à la plus petite valeur entre taille de la mémoire / 4 et 1 Go.

Le parallel collector lève une exception de type OutOfMemoryError si plus de 98% du temps est passé à exécuter le ramasse miettes et que moins de 2% du tas est libéré. Cette sécurité peut être désactivée en utilisant l'option -XX:-UseGCOverheadLimit

45.2.2.2.2. L'utilisation du parallel collector dans la tenured generation

L'algorithme utilisé par le parallel collector dans la tenured generation est identique à celui utilisé par le serial collector (mark-sweep-compact) réalisé par un seul CPU. Ainsi le temps nécessaire à d'éventuels full GC peut donc être long.

45.2.2.2.3. Le choix de l'utilisation du serial collector

Le parallel collector est intéressant pour des applications exécutées sur une machine avec plusieurs CPU n'ayant pas de contrainte forte sur les temps de pause liés au GC (exemple : des applications de type batch).

L'utilisation du parallel collector est intéressante pour des machines disposant de plusieurs processeurs. Le serial collector n'utilise qu'un seul thread pour effectuer ces traitements sur la young generation alors que le parallel collector en utilise plusieurs pour faire les mêmes traitements. Ceci est particulièrement intéressant pour des applications qui utilisent beaucoup de threads et instancient beaucoup d'objets car le temps de traitement du ramasse miettes pour la young generation est réduit.

Depuis la version 5 de Java, le serial collector est choisi par défaut sur des machines de type server. Pour demander l'utilisation du serial collector sur des machines de type client, il faut utiliser l'option -XX:+UseParallelGC

45.2.2.3. Le Parallel Compacting Collector

Le parallel compacting collector est utilisable depuis la version 5 update 6 de Java. Par rapport au parallel collector, le parallel compacting collector utilise un algorithme différent pour le traitement de la tenured generation qui utilise plusieurs threads.

45.2.2.3.1. L'utilisation du parallel compacting collector dans la young generation

L'algorithme utilisé par le parallel compacting collector dans la tenured generation est identique à celui utilisé par le parallel collector.

45.2.2.3.2. L'utilisation du parallel compacting collector dans la tenured generation

L'algorithme utilisé par le parallel compacting collector dans la tenured generation est de type stop the world avec compactage de la mémoire en utilisant plusieurs CPU pour paralléliser ces traitements.

Les traitements du parallel compacting collector comporte trois étapes :

- marking : chaque génération est découpée de façon logique en région de taille fixe : plusieurs threads travaillent en parallèle pour marquer les objets utilisés de chaque région. Pour chaque objet marqué, des informations sont stockées dans la région
- summary : les régions sont vérifiées pour déterminer si elles doivent être compactées selon leur densité de remplissage
- compaction : plusieurs threads déplacent les objets pour remplir les régions qui doivent être compactées afin de compacter les objets du tas

45.2.2.3.3. Le choix de l'utilisation du parallel compacting collector

L'utilisation de cet algorithme est intéressante lorsque la machine dispose de plusieurs CPU. Par rapport du parallel collector, il permet de réduire les temps de pause de l'application.

Pour utiliser le parallel compacting collector, il faut utiliser l'option -XX:+UseParallelOldGC de la JVM.

Le nombre de threads utilisés pour les traitements en parallèle peut être limité en utilisation l'option -XX:ParallelGCThreads=n. Ceci peut être utile notamment sur de gros serveur afin que la JVM ne monopolise pas trop de CPU.

45.2.2.4. Le Concurrent Mark-Sweep (CMS) Collector

De nombreuses applications ont besoin de la meilleure réactivité possible. Généralement, la collection de la young generation est assez rapide. Par contre la collection de la tenured generation peut être assez longue d'autant que cette durée est en relation avec la taille du tas. L'algorithme CMS collector aussi nommé low latency collector permet de réduire la durée d'une collection de la tenured generation en effectuant une partie de ces traitements de façon concurrente avec ceux de l'application.

Le CMS collector a un mode de fonctionnement similaire au serial collector mais certains traitements réalisés dans la tenured generation sont faits dans un ou plusieurs threads dédiés, donc de façon concurrente à l'exécution de l'application. Le but est de réduire les temps de pause de l'application requis pour les collections de la tenured generation.

45.2.2.4.1. L'utilisation du CMS collector dans la young generation

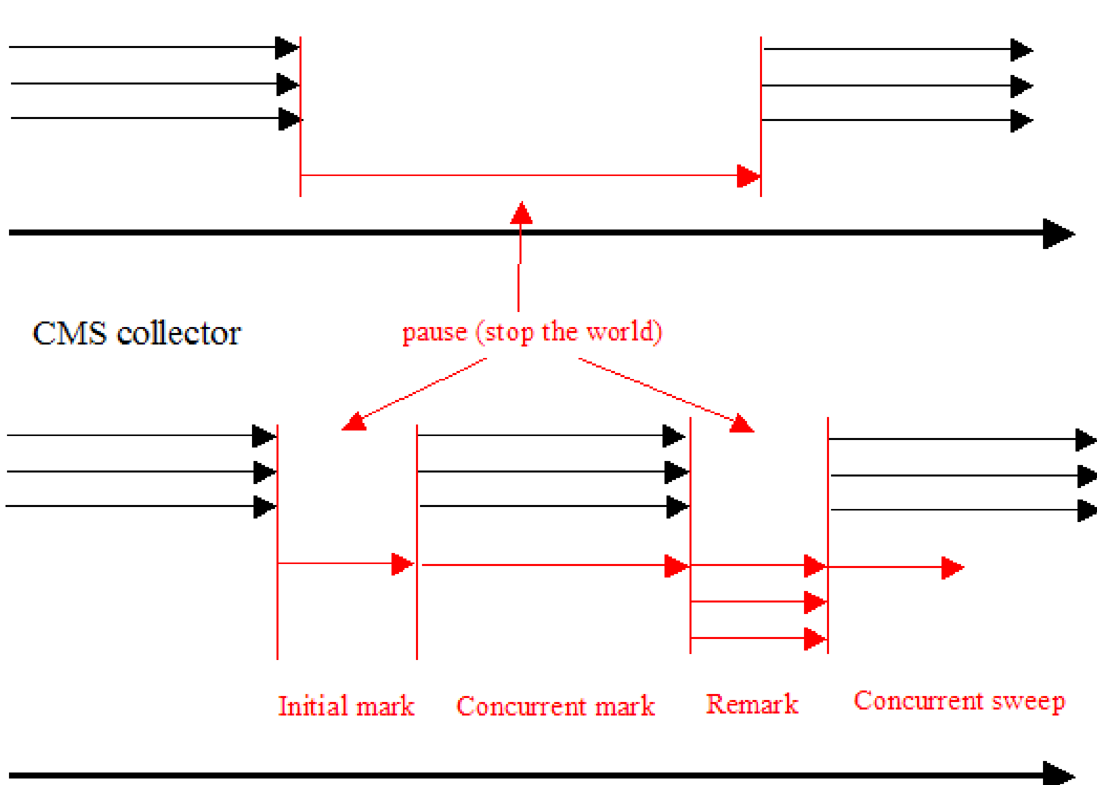
L'algorithme utilisé par le CMS collector dans la young generation est identique à celui utilisé par le parallel collector (plusieurs threads sont utilisés pour réaliser les traitements).

45.2.2.4.2. L'utilisation du CMS collector dans la tenured generation

Une collection réalisée par le CMS collector comporte plusieurs étapes :

- initial mark : cette étape qui met en pause l'application permet de déterminer un ensemble d'objets initiaux utilisés par l'application.
- concurrent marking : cette étape permet de marquer, de façon concurrente à l'exécution de l'application, les objets qui sont utilisés par l'ensemble d'objets initiaux. Comme l'application est en cours d'exécution, les objets vivent et il n'y a donc aucune garantie sur le fait que tous les objets utiles aient été marqués à la fin de cette étape
- remark : cette étape qui met en pause l'application permet de reparcourir les objets modifiés durant l'étape précédente pour s'assurer que tous les objets utiles soient marqués. Les traitements de cette étape sont réalisés par plusieurs threads
- concurrent sweep : toute la mémoire des objets inutiles est récupérée. Les traitements de cette étape sont réalisés par un seul thread
- recalcul de la taille du tas, des statistiques et des données pour la prochaine collection

Le schéma ci-dessous illustre le fonctionnement du serial collector et du CMS collector.



L'étape initial mark est une pause relativement courte. Le temps des traitements concurrents peut être relativement long. Le temps de traitements de la seconde étape provoquant une pause, nommée remark, dépend de l'activité de modification des objets par l'application durant l'étape concurrent mark.

L'exécution concurrente de certains traitements du ramasse miettes avec l'application consomme des ressources CPU qui ne sont pas affectées à cette dernière. Les traitements du ramasse miettes effectués en concurrence sont réalisés avec un seul thread. Sur une machine mono processeur, cela va réduire les performances de l'application tendant à ne pas apporter de gain. Sur une machine bi processeur, un processeur est dédié à l'application, l'autre au thread du ramasse miettes. Plus le nombre de processeurs est important dans la machine, meilleur est le gain en utilisant le CMS Collector.

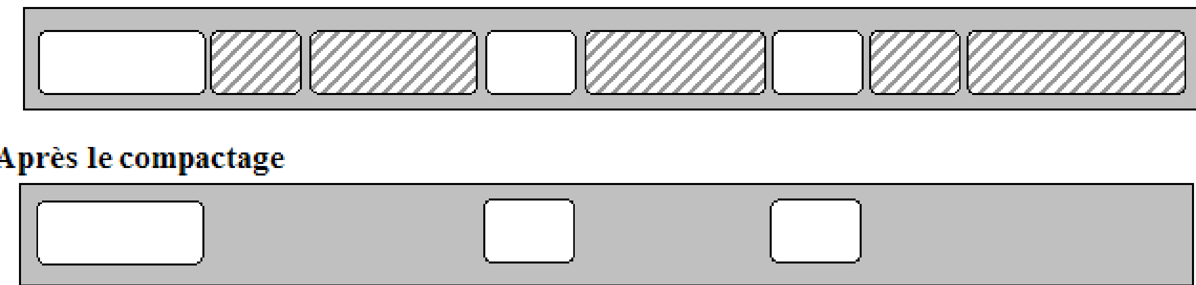
Il est possible de demander l'utilisation du mode incremental dans lequel les traitements réalisés en concurrence avec l'application sont faits de façon incrémentale sur une petite période entre chaque collection sur la young generation. Ceci permet de donner plus de temps de traitement à l'application. Ce mode peut être utile notamment sur des machines ayant

un ou deux processeurs.

Les traitements du CMS Collector dans la tenured generation sont censés être réalisés avant que cette génération ne soient pleine. Il peut arriver que la génération soit remplie avant la fin des traitements : dans ce cas, l'application est mise en pause pour permettre l'exécution complète des traitements du ramasse miettes.

L'exécution de certains traitements en parallèle implique une surcharge de travail pour l'algorithme notamment durant l'étape Remark.

Le CMS collector est le seul algorithme qui ne compacte pas la mémoire après la libération des objets inutilisés. Ceci permet d'économiser du temps de traitements lors des collections mais complexifie l'allocation de mémoire pour de nouveaux objets. Dans le cas d'un compactage, il est facile de connaître le prochain espace mémoire à utiliser puisqu'il correspond à la première adresse mémoire libre de la génération. Sans compactage, il faut gérer une liste des espaces de mémoire disponible (adresse et quantité de mémoire continue). A chaque instanciation, il faut rechercher dans la liste un espace mémoire adéquat et mettre à jour la liste ce qui rend l'instanciation d'un objet dans la tenured generation plus lente. Ce mécanisme nécessite donc aussi plus d'espace mémoire dans la JVM.



Ceci ralentit aussi les traitements de collection sur la young generation puisque la plupart des allocations de mémoire dans la tenured generation sont réalisées par les collections lors de la promotion des objets de la young generation.

De part son mode de fonctionnement, le CMS collector requiert plus de mémoire que les autres collectors pour son propre usage mais aussi parce que l'application peut créer de nouveaux objets pendant l'exécution d'une partie des traitements du ramasse miettes.

Bien que l'algorithme garantisse que tous les objets utilisés soient marqués, il est possible que certains objets marqués soient devenus inutilisés du fait de l'exécution en concurrence de l'application. Dans ce cas, l'espace mémoire de ces objets ne sera pas récupéré durant la collection en cours mais elle le sera à la prochaine collection. L'ensemble de ces objets est nommé floating garbage.

Pour limiter la fragmentation de la mémoire liée au fait que la génération n'est pas compactée, le CMS collector peut fusionner des espaces de mémoires contiguës devenus disponibles.

Contrairement aux autres algorithmes de collections, le CMS collector n'attend que la génération soit pleine pour commencer une collection mais il tente d'anticiper son exécution afin d'éviter que cela ne survienne sinon son temps de traitement serait supérieur à celui d'un serial ou parallel collector. Avec un algorithme de type CMS collector, une collection doit être démarrée de telle sorte que les traitements de la collection soit terminée avant que la old generation soit entièrement remplie.

Le démarrage de la collection peut être lancé selon deux facteurs :

- des statistiques sur le temps d'exécution des précédentes collections et de remplissage de la tenured generation
- à partir d'un certain pourcentage de remplissage de la tenured generation. Ce pourcentage qui par défaut est de 68 peut être modifié en utilisant l'option -XX:CMSInitiatingOccupancyFraction=n

Le CMS collector calcule des statistiques basées sur le fonctionnement de l'application pour tenter d'estimer le temps nécessaire avant que la old generation ne soit remplie et le temps nécessaire aux cycles pour effectuer la collection.

Sur la base des statistiques, les cycles de traitements de la collection sont démarrés avec pour objectif que ceux-ci soient terminés avant que la old generation ne soit pleine.

Il est possible que les estimations provoquent un démarrage trop tardif de la collection ce qui fait rentrer l'algorithme dans un mode nommé concurrent mode failure qui est très couteux car les temps de pauses sont alors beaucoup plus long, ce qui a un effet inverse à celui escompté par l'algorithme sur l'exécution de l'application.

Malheureusement, le calcul de statistiques sur des traitements ayant eu lieu n'est pas toujours le reflet de ce qui se passe ou va se passer. Si de trop nombreux full garbages sont exécutés les uns à la suite des autres, il est possible de modifier plusieurs paramètres pour tenter de remédier au problème :

1. augmenter la valeur du paramètre -XX:CMSIncrementalSafetyFactor
2. augmenter la valeur du paramètre -XX:CMSIncrementalDutyCycleMin
3. désactiver le calcul automatique de la durée des cycles avec l'option -XX:-CMSIncrementalPacing et donner une durée fixe à la durée des cycles de traitements avec - XX:CMSIncrementalDutyCycle

Remarque : ces différentes modifications doivent être faites les unes à la suite des autres, dans l'ordre indiqué jusqu'à ce que le problème disparaisse.

Le traitement de la young generation peut avoir lieu en concurrence avec le traitement de la old generation. Comme le traitement de la young generation est similaire à celui utilisé par le parallel collector en utilisant un algorithme de type stop the world, les threads de traitements de la tenured generation sont interrompus

Les pauses liées à une collection sur la young generation et la old generation sont indépendantes mais peuvent survenir l'une à la suite de l'autre ce qui peut donc allonger le temps de pause de l'application. Pour éviter ce phénomène, l'algorithme tente d'exécuter l'étape remark entre deux pauses liées à la collection de la young generation.

45.2.2.4.3. L'utilisation du mode incrémental

Le mode incremental permet d'utiliser le CMS collector sur des machines avec un seul processeur.

Par défaut, l'algorithme du CMS collector utilise un ou plusieurs threads pour exécuter ses traitements concurrents en dédiant ces threads à ses activités puisque l'algorithme est prévu pour fonctionner sur des machines avec plusieurs CPU.

L'utilisation de cet algorithme peut cependant être intéressante sur des machines avec uniquement un ou deux processeurs. Dans ce cas, l'algorithme propose le mode incremental "i-cms" qui découpe les traitements concurrents de l'algorithme en plusieurs morceaux (duty cycle) exécutés entre les pauses des collections mineures. En dehors de ces cycles, le ou les threads sont suspendus pour permettre au processeur d'exécuter d'autres threads. Le temps d'exécution des cycles est calculé par défaut par l'algorithme en fonction du comportement de l'application dans la JVM (automatic pacing).

Ainsi, le mode incrémental permet de réduire l'impact des traitements concurrents sur l'application en rendant périodiquement la main au processeur pour exécuter l'application. Ces traitements sont découpés en petite unité qui sont exécutées entre deux collections sur la young generation.

L'option -XX:+CMSIncrementalMode permet de demander l'utilisation du mode incrémental : dans ce mode, les traitements réalisés de façon concurrente partagent leur temps d'exécution selon des cycles interrompus par un retour à l'exécution de l'application par le processeur. Le temps d'exécution alloué à un cycle est défini par un pourcentage de temps du processeur accordé pour la collection. Ce pourcentage peut être précisé comme attribut fourni à la JVM ou calculé par l'algorithme en fonction du comportement de l'application

L'option -XX:+CMSIncrementalPacing de la JVM permet de demander de calculer le temps du cycle en fonction du calcul de statistiques issues du comportement de l'application. Par défaut, cette option n'est pas activée en Java 5 et est activée en Java 6.

L'option -XX:CMSIncrementalDutyCycle permet de préciser le pourcentage du temps accordé pour les traitements de l'algorithme entre deux collections mineures. Si l'option -XX:+CMSIncrementalPacing est activée alors cela représente la valeur initiale. La valeur par défaut est 50 en Java 5, ce qui est généralement trop, et 10 en Java 6.

L'option -XX:CMSIncrementalDutyCycleMin permet de préciser le pourcentage du temps minimum accordé pour les traitements de l'algorithme lorsque l'option -XX:+CMSIncrementalPacing est activée. La valeur par défaut est 10 en Java 5 et 0 en Java 6.

Les options recommandées pour i-cms avec Java 5 sont :
-XX:+UseConcMarkSweepGC
-XX:+CMSIncrementalMode
-XX:+CMSIncrementalPacing
-XX:CMSIncrementalDutyCycleMin=0
-XX:CMSIncrementalDutyCycle=10
-XX:+PrintGCDetails
-XX:+PrintGCTimeStamps

Les options recommandées pour i-cms avec Java 6 sont :
-XX:+UseConcMarkSweepGC
-XX:+CMSIncrementalMode
-XX:+PrintGCDetails
-XX:+PrintGCTimeStamps

En fait, ce sont les mêmes options mais la valeur par défaut en Java 6 non précisées est égale à celle recommandée avec Java 5.

45.2.2.4.4. Le choix de l'utilisation du CMS collector

Cet algorithme réduit les temps de pause de l'application liés à son activité en exécutant une partie de celle-ci en concurrence avec l'exécution de l'application. Notamment la recherche initiale des objets utilisés est réalisée dans plusieurs threads. Cet algorithme a toujours besoin de temps de pause mais leur durée est réduite grâce à l'exécution de certains traitements en concurrence avec l'application.

Le CMS collector est prévu pour être utilisé dans une JVM exécutant une application souhaitant avoir de faible temps de pause et qui permette de partager ses ressources processeurs durant son exécution. Généralement, ce sont des applications avec une grosse tenured generation exécutée sur une machine avec plusieurs processeurs. Cet algorithme peut aussi être utilisé pour des applications avec une tenured generation de petite taille, exécutée sur une machine mono processeur avec le mode incrémental activé.

Le CMS collector est recommandé pour des applications qui ont besoins de temps de pause liée au ramasse miettes le plus court possible et qui peuvent accepter d'avoir une partie des traitements du ramasse miettes exécutés en concurrence avec elle. Dans les faits, c'est généralement une application avec une tenured generation de taille importante, exécutée sur une machine avec plusieurs processeurs.

Pour utiliser le CMS collector, il faut utiliser l'option -XX:+UseConcMarkSweepGC.

Les applications qui possèdent de nombreux objets ayant une durée de vie assez longue et qui s'exécute sur une machine multi processeurs peuvent tirer avantage de ce collector : c'est notamment le cas pour les serveurs ou conteneurs web.

Pour utiliser le CMS collector, il faut utiliser l'option -XX:+UseConcMarkSweepGC. Pour demander l'utilisation du mode incremental, il faut utiliser l'option -XX:+CMSIncrementalMode.

45.2.3. L'auto sélection des paramètres du GC par la JVM Hotspot

Les types d'applications qui peuvent être développées en Java et exécutées dans une JVM sont nombreux allant d'une petite applet à une grosse application web ou d'entreprise.

Pour répondre aux besoins variés de ces différents types d'applications, la JVM Hotspot propose plusieurs algorithmes utilisables pour le ramasse miettes. Depuis Java 5, la JVM définit des paramètres par défaut de configuration du ramasse miettes et en fonction du type de machine et du système d'exploitation. Cependant ces valeurs prédéfinies ne sont pas toujours optimales pour une application donnée et il est parfois nécessaire de définir une autre configuration explicitement.

Java 5 propose une fonctionnalité nommée ergonomics dont le but est de configurer certains éléments de la JVM pour permettre d'obtenir de bonnes performances sans configuration. Cette fonctionnalité repose sur un ensemble de règles qui définissent des valeurs par défaut pour :

- la taille du tas
- le ramasse miettes
- le compilateur JIT

Ceci permet d'avoir automatiquement un léger tuning plutôt que d'avoir des valeurs par défaut identiques dans tous les contextes.

La définition de ces valeurs par défaut convient généralement pour des cas standards et elle n'exclue pas d'avoir à configurer soi même ces paramètres pour qu'ils correspondent mieux aux besoins de l'application.

A partir de Java 5, par défaut la JVM détermine plusieurs options de configuration pour le ramasse miettes en fonction de la machine et du système d'exploitation. Ces valeurs par défaut conviennent généralement pour la majorité des applications.

Parmi les valeurs déterminées, il y a le mode de fonctionnement de la JVM HotSpot :

- server : pour une machine avec au moins 2Go de mémoire et plusieurs processeurs sauf pour les machines 32 bits exécutant un système Windows
- client : dans les autres cas

Les valeurs par défaut pour le mode client sont :

- utilisation du serial collector
- taille du tas minimal : 4 Mo
- taille du tas maximal : 64 Mo

Les valeurs par défaut pour le mode server sont :

- utilisation du parallel collector

Quelque soit le mode d'utilisation de la VM, si le parallel collector est utilisé les tailles du tas sont :

- taille du tas minimal : 1/64 de la mémoire physique limité à 1Go
- taille du tas maximal : 1/4 de la mémoire physique limité à 1Go

45.2.4. La sélection explicite d'un algorithme pour le GC

Depuis Java 5, avant de se lancer dans une configuration personnalisée du GC, il faut étudier si la configuration par défaut déterminée par la JVM répond aux besoins. Si ce n'est pas le cas, il est possible de modifier explicitement la configuration.

La meilleure méthodologie pour améliorer les performances est de mesurer, analyser, modifier et d'itérer sur ces trois étapes jusqu'à l'obtention d'un résultat satisfaisant.

Si la configuration par défaut définie par la JVM ne répond pas au besoin, une première amélioration peut être de modifier la taille du tas et des générations qu'il contient. Si cela ne convient toujours pas, il est possible d'essayer un autre algorithme pour le ramasse miettes.

Le choix d'un algorithme pour le ramasse miettes doit prendre en compte plusieurs facteurs, en particulier :

- la taille du tas
- le nombre et la durée de vie des objets
- le nombre de processeurs de la machine

Voici quelques exemples de recommandations :

Conditions	Algorithme recommandé
Si la taille du tas est inférieure à	serial collector -XX:+UseSerialGC

100Mb	
Si la machine est mono processeur	serial collector -XX:+UseSerialGC
Sans contrainte sur les pauses de l'application	parallel collector -XX:+UseParallelGC ou parallel compacting collector -XX:+UseParallelOldGC
Limiter le plus possible les temps de pause de l'application	CMS collector (avec le mode incremental activé si la machine dispose d'un ou deux processeurs) -XX:+UseConcMarkSweepGC

Dans tous les cas, ces recommandations sont à tester pour valider si l'algorithme proposé répond aux besoins de l'application.

Certaines combinaisons d'algorithmes ne sont pas autorisées : dans ce cas la JVM ne démarre pas et affiche un message d'erreur explicite.

Résultat :

```
C:\java\tests>java -XX:+UseConcMarkSweepGC -XX:+UseSerialGC -jar test.jar
Conflicting collector combinations in option list; please refer to the release n
otes for the combinations allowed
Could not create the Java virtual machine.

C:\java\tests>java -XX:+UseConcMarkSweepGC -XX:+UseParallelGC -jar test.jar
Conflicting collector combinations in option list; please refer to the release n
otes for the combinations allowed
Could not create the Java virtual machine.
```

45.2.5. Demander l'exécution d'une collection majeure.

Généralement l'exécution du ramasse miettes est conditionnée par un manque d'espace libre. Cela permet dans un premier temps de libérer de l'espace. Si cela ne suffit pas alors l'espace mémoire est agrandit jusqu'à atteindre le maximum défini.

La méthode gc() de la classe System permet de demander l'exécution du ramasse miettes.

Cependant, le moment d'exécution du ramasse miettes n'est pas facilement prédictible. Même l'appel à la méthode gc() de la classe System n'implique pas obligatoirement l'exécution du ramasse miettes mais sollicite simplement une demande d'exécution.

Avec certains algorithmes du ramasse miettes, forcer l'invocation du ramasse miettes pour être préjudiciable sur les performances notamment pour de grosses applications. L'option -XX:+DisableExplicitGC de la JVM demande à la JVM d'ignorer les demandes explicites d'exécution du ramasse miettes.

45.2.6. La méthode finalize()

Les Java Langage Specifications impose au ramasse miettes d'appeler la méthode de finalisation de l'objet héritée de la classe Object avant de libérer la mémoire. Ainsi, le ramasse miettes a l'obligation par ces spécifications d'invoquer la méthode finalize() lorsque l'espace mémoire d'une instance d'un objet va être récupéré.

Il est alors facile d'imaginer de mettre des traitements de libération de ressources par exemple dans cette méthode puisque la JVM garantie l'appel de cette méthode lorsque l'objet n'est plus utilisé.

Malheureusement, seule l'invocation est garantie : l'exécution de cette méthode dans son intégralité ne l'est pas notamment si une exception est levée durant ses traitements.

De plus, le moment ou l'espace mémoire va être récupéré et donc le moment où la méthode finalize() sera invoquée n'est

pas prédictible.

Pour ces deux raisons, il ne faut surtout pas utiliser la méthode finalize() pour libérer des ressources.

Généralement une bonne pratique est de ne pas faire usage dans la mesure du possible de la méthode finalize() et de ne surtout pas utiliser la garbage collector pour faire autre chose que la libération de la mémoire.

45.3. Le paramétrage du ramasse miettes de la JVM HotSpot

La JVM Hotpsot de Sun propose de nombreux paramètres relatifs à l'activité du ramasse miettes.

45.3.1. Les options pour configurer le ramasse miettes

La JVM propose de nombreuses options pour configurer le ramasse miettes : elles permettent notamment de modifier la taille du tas et des générations, choisir un algorithme, le configurer et obtenir des informations sur son exécution.

Plusieurs paramètres permettent de configurer la taille des différents espaces mémoire de la JVM.

L'allocation de la mémoire pour la JVM se fait au moyen de plusieurs options préfixées par -X

Option	Rôle
-Xms	Taille initiale du tas (heap)
-Xmx	Taille maximale du tas (heap)
-Xmn	Taille de la young generation du tas
-Xss	Taille de la pile (stack) de chaque thread. Si celle si trop petite, une exception de type StackOverFlowError est levée Exemple : `-Xss1024k`

Remarque : sur certaines machines utilisant Linux, il est parfois nécessaire de modifier la taille de la pile au niveau des paramètres du système d'exploitation en utilisant la commande ulimit -s.

Plusieurs paramètres non standards sont proposés par la JVM Hotspot pour gérer la taille du tas et des générations.

Option	Description
-XX:MinHeapFreeRatio=*n*	Définir le ratio minimum d'espace libre du tas avant que sa taille ne soit agrandie. Exemple : si le ratio est de 20 et que le pourcent d'espace libre n'est plus que de 20% alors la taille du tas est agrandie pour qu'il y ait 20% d'espace libre.
-XX:MaxHeapFreeRatio=*n*	Définir le ratio maximum d'espace libre du tas avant que sa taille ne soit réduite. Exemple : si le ratio est de 70 et que le pourcent d'espace libre dépasse les 70% alors la taille du tas est réduite pour qu'il y est 70% d'espace libre.
-XX:NewSize=*n*	Définir la taille initiale de la young generation.
-XX:NewRatio=*n*	Définir le ratio entre la young generation et la old generation. Exemple : Si n vaut 3 alors le ratio est de 1:3. Ainsi la taille de la young generation est de un quart de la taille du tas.

-XX:SurvivorRatio=*n*	Définir le ratio entre les espaces survivor et eden dans la young generation Exemple : Si n vaut 3 alors le ratio est de 1:3. Ainsi la taille de la young generation est de un quart de la taille du tas. Exemple : si n vaut 6 alors le ratio est 1:8 de la young generation (ce n'est pas 1:7 car il y a deux espaces survivor).
-XX:MaxPermSize=*n*	Définir la taille maximale de la permanent generation

Il est tout à fait normal que la taille de la JVM observée sur le système soit supérieure à la taille fournie par l'option -Xmx. La valeur de cette option ne concerne que le tas et non tout l'espace mémoire de la JVM qui inclus aussi entre autre les piles (stack), la permanente generation, ...

Les valeurs à affecter aux options -Xms et -Xmx dépendent de l'application exécutée dans la JVM. La taille minimale doit supporter l'espace requis par l'application. Sur un poste utilisateur il est généralement préférable de mettre des valeurs minimales et maximales différentes pour limiter la consommation de mémoire sur la machine tout en lui permettant de grossir aux besoins. Sur un serveur, il est généralement préférable de mettre la même valeur car les ressources mémoire sont généralement moins limitées et cela évite les allocations de mémoire successives.

Les options pour choisir l'algorithme utilisés par le ramasse miettes sont :

Option	Algorithme utilisé par le ramasse miettes
-XX:+UseSerialGC	Serial collector
-XX:+UseParallelGC	Parallel collector
-XX:+UseParallelOldGC	Parallel compacting collector
-XX:+UseConcMarkSweepGC	Concurrent mark-sweep collector (CMS collector)

Les options pour afficher des informations sur l'exécution du ramasse miettes sont :

Option	Description
-XX:+PrintGC	Afficher des informations de base à chaque exécution du ramasse miettes
-XX:+PrintGCDetails	Afficher des informations détaillées à chaque exécution du ramasse miettes
-XX:+PrintGCTimeStamps	Afficher la date-heure de début d'exécution du ramasse miettes

Le paramètre -verbose:gc permet aussi d'afficher dans la console des informations sur chaque collecte effectuée par le ramasse miettes.

Les options pour les algorithmes parallel et parallel compacting collector sont :

Option	Description
-XX:ParallelGCThreads=*n*	Nombre de threads utilisé par le ramasse miettes (par défaut, c'est le nombre de CPU de la machine)
-XX:MaxGCPauseMillis=*n*	Demander au ramasse miettes d'essayer de limiter son temps d'exécution à celui fourni en millisecondes en paramètre.
-XX:GCTimeRatio=*n*	Demander au ramasse miettes d'essayer de respecter un ratio 1/(1+n) du temps total utilisé pour les activités du ramasse miettes. La valeur par défaut est 99

Les principales options pour l'algorithme CMS collector sont :

Option	Description
-XX:+CMSIncrementalMode	Activer le mode d'exécution des traitements concurrents de façon incrémentale
-XX:+CMSIncrementalPacing	Activer le calcul automatique de statistiques pour déterminer le temps de traitement du ramasse miettes dans le mode incrémental.
-XX:ParallelGCThreads=*n*	Nombre de threads utilisés par le ramasse miettes pour le traitement de la young generation et des traitements concurrents de la old generation.

45.3.2. La configuration de la JVM pour améliorer les performances du GC

Le ramasse miettes peut induire de véritable problème de performance pour certaines applications en fonction des besoins de celle-ci et du paramétrage de la JVM.

En cas de problème de performance avec le ramasse miettes, si la taille du tas doit être modifiée, il faut aussi généralement adapter la taille de chacune des générations.

En cas de fuite de mémoire ou d'inadéquation de la taille du tas avec les besoins de l'application, il est possible que les performances de la JVM se dégradent très fortement car elle peut occuper une large partie de ces traitements à l'exécution du ramasses miettes de façon répétée.

Une bonne adéquation entre les besoins de l'application et la configuration de la JVM peut permettre de réduire le temps nécessaire à l'exécution du ramasse miettes durant l'exécution de l'application.

Plusieurs indicateurs peuvent être utilisés pour mesurer les performances du ramasse miettes :

- throughput : pourcentage du temps consacré à l'exécution de l'application par la JVM
- pause : temps d'arrêt de l'application lié à l'exécution du ramasse miettes
- footprint : espace mémoire consommé par la JVM
- promptness : temps entre le moment ou un objet n'est plus utilisé et le moment ou son espace mémoire est libéré

Il n'y a pas de règle absolue pour optimiser les performances du ramasse miettes : cette optimisation est dépendante de l'application exécutée dans la JVM. Par exemple, dans une application standalone, il est très important d'avoir l'indicateur pause le plus court possible alors qu'il n'est généralement pas crucial pour une application de type web.

Les besoins relatifs aux performances du ramasse miettes dépendent de la typologie d'applications exécutés, par exemple :

- dans une application standalone, les temps de pause doivent être réduits au maximum
- dans une application web, c'est le thoughput qui est important dans la mesure où les pauses peuvent être masquées par les temps de latence du réseau

Cette optimisation doit tenir compte des priorités données à chaque indicateurs.

Par exemple, plus la taille de la young generation est importante, le thoughput s'améliore mais l'empreinte mémoire et les temps de pause augmente.

A l'inverse, plus la taille de la young generation est petite plus le thoughput est bon

Il n'y a donc pas une façon unique d'optimiser la taille des générations mais elle doit utiliser les besoins et requis de l'application.

Plusieurs paramètres peuvent avoir une influence sur la taille des générations. Au lancement de la JVM, l'espace entier défini par le paramètre -Xmx du tas est réservé. Si la valeur du paramètre -Xms est plus petite que celle de -Xmx alors

seule la quantité indiquée par -Xms est immédiatement disponible pour le tas. Le reste de ma mémoire est dite virtuelle : elle sera utilisée au besoin.

Les différentes générations (young, tenured, permanent) peuvent ainsi grossir jusqu'à atteindre leur taille maximale respective. Certaines caractéristiques sont fournies sous la forme de ratio. Par exemple, le paramètre NewRatio définit la proportion de la taille de la young et tenured generation dans le tas.

La quantité de mémoire du tas gérée est un facteur important pour les performances du ramasse miettes.

Par défaut, les traitements du ramasse miettes peuvent faire grossir ou réduire la taille du tas lors de chaque collection afin de respecter la quantité de mémoire libre souhaitée précisée sous la forme de pourcentage par les paramètres -XX:MinHeapFreeRatio=<minimum> and -XX:MaxHeapFreeRatio=<maximum>

Attention : l'augmentation de la taille de mémoire du tas provoque généralement des effets de bord sur les temps de traitements liés à l'exécution du garbage collector notamment parce que ce dernier est invoqué moins fréquemment mais ses temps de traitement sont plus longs.

Donner la même valeur au paramètre -Xms et -Xmx permet d'éviter à la JVM de devoir effectuer des calculs sur la taille des différentes régions mais cela empêche aussi la JVM de procéder à des ajustements si les valeurs fournies ne sont pas judicieuses.

Le ratio entre la young generation et la tenured generation dans le tas est un facteur important dans les performances du ramasse miettes. Plus le ratio de la young generation est important, moins il y a de collections mineures qui sont effectuées. Par contre, cela implique une taille de la tenured generation plus importante et donc un accroissement du nombre potentiel de collections majeures. La valeur du ratio entre les deux générations dépend donc du nombre d'objets créés et de la durée de vie de ces objets dans l'application.

La taille de la young generation est déterminée par le paramètre NewRatio qui indique le ratio de la young et de la tenured generation dans le tas.

La taille de la young generation peut être précisée avec le paramètre -XX:NewRatio=n où n représente le ratio entre la young generation et la old generation.

Exemple : si n vaut 3, la young generation aura une taille de 1/4 de la taille totale du tas.

Le paramètre -XX:NewSize permet de préciser la taille initiale de la young generation.

Le paramètre -XX:MaxNewSize permet de préciser la taille maximale de la young generation.

La valeur par défaut de l'attribut NewRatio est dépendante de la plate-forme et du mode d'exécution de la VM Hotspot (client ou server).

La définition de la taille des générations du tas peut se faire de plusieurs façons :

- en utilisant l'attribut Newratio : pour avoir une définition sous la forme de ratio
- en utilisant les attributs NewSize et MaxNewSize pour avec une définition précise

Remarque : l'attribut -Xmn est un raccourci pour NewSize

Plus la taille de la young generation est importante plus les chances qu'un objet meurt dans la young generation et ne soit donc pas promu dans la old generation est élevé. Cependant, il n'est pas recommandé d'avoir une taille très importante pour la young generation car cela fera exécuter le ramasse miettes moins souvent mais son temps d'exécution sera plus long et comme ces traitements sont de type stop the world, les temps de pause de l'application seront plus long. L'idéale est d'adapter la taille de la génération en fonction de l'application exécutée dans la JVM.

Le paramètre -XX:SurvivorRatio=n permet de préciser le ratio entre l'espace eden et les deux espaces survivor dans la young generation.

Exemple : si n vaut 6 alors le ratio est 1:6. Dans ce cas, chaque espace survivor occupera 1/8 de la taille de la young generation (ce n'est pas 1/7 car il y a deux espaces de type survivor)

Si l'espace requit pour copier un objet dans l'espace survivor n'est pas assez grand, alors l'objet est promu directement dans la old generation.

A chaque collection mineure, la ramasse miettes détermine le nombre de collections qu'un objet doit avoir subit avant d'être promu dans la old generation. Ce nombre est déterminé de façon à ce qu'à la fin de la collection l'espace survivor soit à moitié rempli

L'option -XX:+PrintTenuringDistribution permet de voir la répartition par âges des objets de la young generation. Ceci permet de voir la répartition de la durée des objets de la young generation.

Il faut tout d'abord décider de la quantité de mémoire qui sera affecté au tas. Ensuite, il est possible de mesurer les performances et d'ajuster la taille de la young generation en fonction du comportement de l'application.

Pour la plupart des applications, la permanent generation n'influe pas de façon importante sur les performances du ramasse miettes. Pour des applications qui chargent et/ou génèrent beaucoup de classes, il faut augmenter la taille de cette génération pour éviter des manques de mémoire.

La taille du tas ne permet pas à elle seule de déterminer la quantité de mémoire utilisée par le processus système de la JVM puisque le tas ne représente qu'une partie de la mémoire de la JVM.

Il ne doit donc pas être étonnant que la quantité de mémoire affichée par le système (TaskManager sous Windows ou top sous Unix like par exemple) soit supérieure à la taille maximale du tas précisée avec l'option -Xmx.

45.4. Le monitoring de l'activité du ramasse miettes

La suite de cette section sera développée dans une version future de ce document

45.5. Les différents types de référence

Java 1.2 propose plusieurs types de référence qui contiennent une référence particulière sur un objet.

Ces références sont définies dans des classes du package java.lang.ref :

- Soft Reference
- Weak Reference
- Hard Reference
- Phantom Reference

Ces différentes références peuvent être utilisées par le ramasse miettes pour récupérer de la mémoire au cas où celle ci commence à manquer.

La suite de cette section sera développée dans une version future de ce document

45.6. L'obtention d'informations sur la mémoire de la JVM

La classe Runtime propose deux méthodes pour obtenir des informations basiques sur la mémoire occupée par le tas :

- totalMemory() : renvoie la quantité totale de mémoire du tas
- freeMemory() : renvoie la quantité de mémoire libre du tas

Depuis la version 5 de la JVM, il est aussi possible d'obtenir des informations sur la mémoire en utilisant les MBeans JMX exposés par la JVM.

45.7. Les fuites de mémoire (Memory leak)

La libération de la mémoire des objets inutilisés est implicite en Java grace au ramasse miettes alors qu'elle est explicite dans d'autres langages (en Pascal avec l'instruction dispose, en C avec l'instruction free, ...). Ceci facilite le travail du développeur puisqu'il n'a pas à libérer explicitement la mémoire des objets.

Il est facile de penser que la libération de mémoire étant assurée par le garbage collector de la JVM, le développeur n'a plus aucune responsabilité à ce sujet et les fuites de mémoires sont impossibles. Ce raisonnement est le résultat de la méconnaissance du mode de fonctionnement du garbage collector.

Le ramasse miettes doit s'assurer pour libérer la mémoire d'un objet que celui n'est plus utilisé : pour le déterminer, il recherche s'il existe parmi les objets de la JVM, une référence vers l'objet. Même s'il n'est plus utilisé mais qu'il existe encore une référence sur l'objet, la mémoire de celui-ci n'est pas libérée. Ceci rend la tâche de détection d'une fuite de mémoire particulièrement difficile car il est facile de savoir si un objet est encore utile mais il est difficile de savoir s'il existe encore une référence vers l'objet.

Une mauvaise utilisation de l'API collection par exemple peut notamment favoriser les fuites de mémoires.

Une fuite de mémoire se traduit généralement par une augmentation de la taille du heap pouvant aller jusqu'à un arrêt de la JVM avec une exception de type OutOfMemoryError.

Le premier réflexe lorsqu'une exception de type OutOfMemoryError est levée concernant le tas est d'augmenter la taille de la mémoire de JVM. Cependant, si cette erreur est liée à une fuite de mémoire cela ne fait que reporter sa levée.

L'indicateur le plus visible lors d'une possible fuite de mémoire est la levée d'une exception de type OutOfMemory. Mais la levée de cette exception n'implique pas obligatoirement une fuite mémoire mais peut être simplement un manque de mémoire pour permettre l'exécution de l'application.

Cependant, c'est l'issue fatale suite à une fuite de mémoire qui peut être plus ou moins longue. Ceci est particulièrement vrai pour des applications serveurs car elles ne sont généralement pas redémarrées fréquemment.

Même si c'est un travail long et difficile, il faut toujours traiter une fuite de mémoire avec une grande attention. La solution n'est pas d'augmenter la taille du tas car cela ne fera que reporter l'échéance fatale. La solution n'est pas non plus de redémarrer périodiquement la JVM car généralement les applications concernées doivent avoir un taux de disponibilité élevé.

La suite de cette section sera développée dans une version future de ce document

45.7.1. Diagnostiquer une fuite de mémoire

Le diagnostique d'une fuite de mémoire dans une application Java est une tâche difficile et longue qui nécessite généralement des outils qui vont permettre de voir quels sont les objets stockés dans la mémoire de la JVM.

Le JDK fournit de plus en plus d'outils pour effectuer ces recherches et donner des indications sur l'origine du problème mais ils sont en ligne de commande et sont donc peu productifs notamment jmap, jhat et jconsole. Leur intérêt est cependant d'être fourni en standard. A partir de Java 6 update 7 l'outil graphique Visual VM propose de regrouper ces fonctionnalités de façon conviviale.

Le paramètre -XX:+HeapDumpOnOutOfMemoryError de la JVM HotSpot permet de demander la création d'un dump du tas au cas où l'exception de type OutOfMemoryError est levée. La commande jmap permet alors de fournir un histogramme en utilisant l'option -histo suivi du fichier contenant le dump.

Le paramètre -XX:+PrintClassHistogram de la JVM HotSpot permet de demander l'affichage dans la console de l'histogramme des classes du tas lorsque la combinaison Ctrl + Arret défil est utilisée.

Il existe aussi plusieurs outils de profiling open source (Eclipse TPTP, Eclipse Mat, Netbeans Profiler, ...) ou commerciaux.

Bien que leur utilisation facilite le travail, la détection d'une fuite de mémoire est souvent délicate car :

- le tas d'une JVM contient généralement de très nombreux objets : par exemple une petite application peut facilement avoir plusieurs milliers d'objets en mémoire
- l'analyse des objets contenus dans le tas nécessite une bonne connaissance de l'application
- généralement la fuite est légère
- l'analyse de la liste des objets et du nombre de leur instance est le principal indicateur pour déterminer l'origine de la fuite

La détection d'une fuite de mémoire n'est souvent qu'une hypothèse en cas d'arrêt de la JVM par manque de mémoire. Dans ce cas, la suspicion de fuite de mémoire est conditionnée par l'importance de la fuite, la taille de la mémoire de la JVM et la durée de vie de la JVM.

Exemple : une petite fuite de mémoire dans une application de type client lourd fréquemment relancée ne sera peut être jamais détectée par contre une petite fuite de mémoire dans une application de type web dont la durée de vie est longue a des chances de provoquer tôt ou tard une erreur de type OutOfMemoryError. L'inverse est vrai aussi.

Le grand avantage des fuites de mémoire en Java est qu'elles n'ont pas d'impact sur le système d'exploitation. La JVM et donc l'application s'arrête et la mémoire qui lui était allouée est restituée au système.

Le simple fait de regarder, via les outils du système d'exploitation, la quantité de mémoire consommée par la JVM ne peut en aucun cas fournir d'indication sur une possible fuite de mémoire dans l'application.

La quantité de mémoire indiquée par le système ne contient qu'une partie relative au tas de la JVM. De plus la taille du tas peut varier entre le minimum et le maximum défini. Au démarrage de la VM, l'espace de mémoire du tas alloué correspond à la valeur minimale. Au fur et à mesure des besoins, la taille du tas peut grossir jusqu'à la valeur maximale et cela sans attendre la fin des traitements du ramasse miettes. Dans ce cas, la quantité de mémoire indiquée par le système grossit mais n'implique pas obligatoirement une fuite de mémoire.

Un outil de type profiler est nécessaire pour permettre d'inspecter le contenu du tas pour connaître le nombre d'objets, le nombre d'instances de chaque classe, ...

Même avec ce type d'outil, la recherche d'une fuite de mémoire est un processus généralement long et itératif.

Certaines entités sont propices à la génération de fuite de mémoire :

- utilisation de collections qui ont une durée de vie assez longue (par exemple les collections déclarées static).
- Dans une application graphique, abonnement à un listener et oublie de se désabonner
- ...

Les conséquences d'une fuite de mémoire dans une application Java sont généralement moins dramatiques que dans des applications natives dans la mesure où en Java seule la JVM et donc l'application risque de s'arrêter. Une fuite de

mémoire peut engendrer un arrêt de la JVM dans laquelle l'application s'exécute mais le système d'exploitation reste opérationnel. Dans une application native; une fuite de mémoire peut aller jusqu'à nécessiter le redémarrage du système d'exploitation si ce dernier n'a plus assez de mémoire.

Les fuites de mémoire dans une application peuvent avoir plusieurs origines dont les plus communes sont :

- la présence de référence sur des objets non désirée ou non connue
- l'oublie de la libération de certaines ressources externes
- bug dans une bibliothèque tierce

Le ramasse miettes invoque la méthode finalize() si celle-ci est implémentée pour un objet avant que son espace mémoire ne soit récupéré. Il n'y a cependant aucune garantie sur la bonne exécution de cette méthode : il ne faut surtout s'en servir pour libérer de ressources sous prétexte qu'elle est invoquée automatiquement par le ramasse miettes.

45.8. Les exceptions liées à un manque de mémoire

Deux exceptions différentes peuvent être levées par la JVM selon l'origine du manque de mémoire StackOverFlowError et OutOfMemoryError. Ces deux exceptions ne sont pas checkées mais provoquent un arrêt de la JVM si elles ne sont pas traitées dans un bloc catch durant la remontée de la pile d'appels du thread.

45.8.1. L'exception de type StackOverFlowError

Si la taille de la pile est trop petite alors une exception de type java.lang.StackOverFlowError est levée.

Il y a deux grandes origines si la taille de la pile n'est pas assez importante :

- généralement c'est un appel récursif à une méthode sans condition d'arrêt (une méthode qui s'appelle elle-même)
- la quantité de données à stocker dans la pile est supérieure à la taille de la pile : dans ce cas il faut agrandir la taille des piles en utilisant l'option -Xss. Attention cependant car cette option s'applique à toutes les piles (une par thread)

45.8.2. L'exception de type OutOfMemoryError

Une exception de type OutOfMemoryError est assez courante lors de l'exécution d'applications Java : elle indique qu'il n'y a pas l'espace disponible pour créer de nouveaux objets même après l'exécution du ramasse miettes et qu'il n'y a pas la possibilité d'agrandir la taille du tas.

L'exception OutOfMemoryError peut concerner plusieurs parties de la mémoire de la JVM : cette partie est explicitement indiquée dans le message de l'exception :

- java.lang.OutOfMemoryError: Java heap space
- java.lang.OutOfMemoryError: PermGen space
- java.lang.OutOfMemoryError: Requested array size exceeds VM limit
- java.lang.OutOfMemoryError: Request <size> bytes for <reason>. Out of swap space?.
- java.lang.OutOfMemoryError: <reason> <stack trace> (Native method)

Il est donc important de bien prendre en compte le message de l'exception OutOfMemoryError pour pouvoir y apporter une solution.

Une exception de type OutOfMemoryError n'est pas obligatoirement un problème de fuite de mémoire mais simplement une mauvaise adaptation de la configuration de la JVM aux besoins de l'application.

S'il est nécessaire de réduire l'empreinte mémoire de l'application dans la JVM, il faut tenter de réduire le nombre d'objets ou de limiter la durée de vie de certains objets, par exemple :

* Essayer de réduire la durée de vie de certains objets, par exemple réduire le timeout des sessions http d'un conteneur web
* Si l'application utilise un cache alors il faut vérifier sa taille et la limiter. Il est aussi possible d'utiliser des objets ayant des soft références dans les caches ce qui permettra au ramasse miettes de supprimer ces objets en cas de manque d'espace dans le tas
* S'assurer de la correcte libération de ressources externes
* ...

L'utilisation d'un outil de profiling peut être nécessaire voir obligatoire pour analyser le contenu de la mémoire de la JVM et déterminer l'origine de la consommation mémoire. L'utilisation de ce type d'outils induit forcement un overhead et réduit donc sensiblement les performances. Leur utilisation doit donc être limitée dans un environnement de production

45.8.2.1. L'exception OutOfMemoryError : Java heap space

Une exception de type OutOfMemory est levée avec le message "Java heap space" lorsque l'espace mémoire libre du tas (heap) ne permet plus la création de nouveaux objets malgré l'exécution du ramasse miettes.

Dans ce cas, l'exception peut avoir plusieurs origines :

* l'espace mémoire allouée au tas de la JVM est insuffisant pour créer les objets requis par l'application. C'est généralement le cas pour des applications gourmandes en ressources (grosses applications web ou graphiques, ...)
* une fuite de mémoire empêche le ramasse miettes de libérer des objets qui sont pourtant inutilisés mais dont il existe encore des références. Ainsi ces objets ne sont jamais libérés et occupent de plus en plus d'espace dans le tas jusqu'à occuper tout l'espace disponible.
* de nombreux objets possèdent un finalizer. Lors de la prise en compte de ces objets par le ramasse miettes, ces objets sont mis dans une file pour être ultérieurement traités par un thread dédié qui va exécuter le finalizer avant le libérer la mémoire
* ...

45.8.2.2. L'exception OutOfMemoryError : PermGen space

Une exception de type OutOfMemory est levée avec le message "PermGen space" lorsque l'espace mémoire alloué à la permanent generation n'est pas assez important pour contenir toutes les méta données utilisées par la JVM.

C'est généralement pour des applications côté serveur car elles s'exécutent dans un même conteneur et utilisent généralement de nombreuses classes différentes liés à l'utilisation de plusieurs frameworks.

La représentation interne des chaînes de caractères de type constante est aussi stockée dans un pool de la permanent generation. Lorsque la méthode intern() de la classe String est invoquée, elle recherche dans le pool si la chaîne existe déjà. Si c'est le cas, elle renvoie celle du pool sinon elle l'ajoute. L'espace mémoire requis pour le permanent generation est donc plus imporant lorsqu'une application utilise beaucoup de chaînes de caractères sous la forme de constantes.

La seule solution est alors d'agrandir l'espace mémoire allouée à la permanent generation, par exemple en utilisant l'option -XX:MaxPermSize pour une JVM Hotspot.

45.8.2.3. L'exception OutOfMemoryError : Requested array size exceeds VM limit

Lorsqu'une exception de type OutOfMemory est levée avec le message "Requested array size exceeds VM limit" lorsqu'une tentative de création d'un tableau qui requiert plus de mémoire que l'espace mémoire libre du tas.

Si la taille du tableau à créer est normale alors la seule solution est d'augmenter la taille du tas de la JVM.

46. La décompilation et l'obfuscation

Le compilateur transforme un fichier source en fichier de classe contenant du byte code. Ce byte code est ensuite lu est interprété par la JVM.

La décompilation consiste à générer du code source à partir du bytecode pour effectuer un reverse engineering. Un des outils pionnier dans cette activité est Mocha qui a fait couler beaucoup d'encre.

L'obfuscation consiste à rendre le résultat d'une décompilation difficilement lisible voir impossible.

46.1. Décompiler du byte code

La décompilation consiste à produire un fichier source Java à partir d'un fichier de classe contenant du byte code. C'est l'opération inverse de la compilation. Ce processus est possible car le byte code est standardisé et parfaitement documenté.

Attention : ce processus est généralement prohibé pour du code dont on n'est pas l'auteur ou qui ne soit pas open source. Avant de réaliser une décompilation, il est important de se renseigner sur la licence du code qui va subir cette opération afin de ne pas enfreindre la licence d'utilisation.

La décompilation est possible parce que la compilation du code source ne produit pas du code machine binaire mais produit du byte code qui est un langage indépendant de toute plate-forme. Lors de son exécution, le byte code peut être interprété ou compilé en code machine. Le format du byte code est assez proche du code source, ce qui permet de réaliser une décompilation relativement facilement notamment pour ce qui concerne la logique des traitements.

Il existe plusieurs outils pour décompiler du byte code

Outils	Url
JReversePro	http://jrevpro.sourceforge.net/
Jad (the fast Java Decompiler)	http://www.kpdus.com/jad.html
DJ Java decompiler (utilise Jad)	http://members.fortunecity.com/neshkov/dj.html
IdeaJad (utilise Jad)	http://www.tagtraum.com/ideajad.html
JODE	http://jode.sourceforge.net/
CafeBabe	http://www.geocities.com/CapeCanaveral/Hall/2334/Programs/cafebabe.html

46.1.1. JAD : the fast Java Decompiler

Jad est un décompilateur gratuit pour un usage non commercial ou personnel qui est particulièrement efficace et véloce car il est écrit en C++.

Il faut télécharger le fichier jadnt158.zip à l'url http://www.kpdus.com/jad.html et décompresser l'archive dans un répertoire du système. Le plus simple est d'ajouter ce répertoire au Path du système.

La classe ci-dessous est utilisée comme exemple

Exemple :

```
package com.jmdoudoux.test;

public class MaClasse {

  /**
   * @param args
   */
  public static void main(
      String[] args) {
    System.out.println("Bonjour");
  }

}
```

Exécuter jad en lui passant en paramètre le nom du fichier .class à décompiler

Exemple :

```
C:\Documents and Settings\jmd\workspace\Tests\bin\com\jmdoudoux\test>jad MaClass
e.class
Parsing MaClasse.class... Generating MaClasse.jad
```

L'exécution produit un fichier MaClasse.jad

Exemple :

```
// Decompiled by Jad v1.5.8g. Copyright 2001 Pavel Kouznetsov.
// Jad home page: http://www.kpdus.com/jad.html
// Decompiler options: packimports(3)
// Source File Name:   MaClasse.java

package com.jmdoudoux.test;

import java.io.PrintStream;

public class MaClasse
{

    public MaClasse()
    {
    }

    public static void main(String args[])
    {
        System.out.println("Bonjour");
    }
}
```

46.1.2. La mise en oeuvre et les limites de la décompilation

Cette section va utiliser la classe ci dessous

Exemple :

```
package com.jmdoudoux.test.decompile;
```

```
import java.util.ArrayList;
import java.util.Date;
import java.util.List;

/**
 * Classe de test
 *
 */
public class MaClasse {

  private String nom;
  protected String prenom;
  public Date dateNaissance;
  public List commandes = new ArrayList();

  public static void main(
      String[] args) {
    MaClasse maClasse = new MaClasse("nom1","prenom1",new Date());
    maClasse.ajouterCommande("commande 1");
    maClasse.ajouterCommande("commande 2");
    System.out.println(maClasse);
  }

  /**
   * Constructeur
   * @param nom
   * @param prenom
   * @param dateNaissance
   */
  public MaClasse(String nom, String prenom, Date dateNaissance) {
    super();
    this.nom = nom;
    this.prenom = prenom;
    this.dateNaissance = dateNaissance;
  }

  /**
   * Ajouter une commande
   * @param libelle libelle de la commande
   */
  public void ajouterCommande(String libelle) {
    commandes.add(libelle);
  }

  @Override
  public String toString() {
    StringBuilder sb = new StringBuilder("");
    sb.append("Nom : ");
    sb.append(nom);
    sb.append("\n");
    sb.append("Prenom : ");
    sb.append(prenom);
    sb.append("\n");
    sb.append("Date de naissance : ");
    sb.append(dateNaissance);
    sb.append("\n");
    sb.append("Commandes :\n");
    for(Object commande : commandes) {
      sb.append("  ");
      sb.append(commande);
      sb.append("\n");
    }

    return sb.toString();
  }
}
```

Exemple : décompilation avec jad

Exemple :

```
C:\java\tests>javac com/jmdoudoux/test/decompile/MaClasse.java
Note: com\jmdoudoux\test\decompile\MaClasse.java uses unchecked or unsafe operat
ions.
Note: Recompile with -Xlint:unchecked for details.

C:\java\tests>cd com/jmdoudoux/test/decompile

C:\java\tests\com\jmdoudoux\test\decompile>dir
 Volume in drive C has no label.
 Volume Serial Number is 1F23-7A9

 Directory of C:\java\tests\com\jmdoudoux\test\decompile

01/04/2008  10:15    <DIR>          .
01/04/2008  10:15    <DIR>          ..
01/04/2008  10:15             1 755 MaClasse.class
01/04/2008  09:58             1 545 MaClasse.java
               2 File(s)          3 300 bytes
               2 Dir(s)  57 852 260 352 bytes free

C:\java\tests\com\jmdoudoux\test\decompile>jad MaClasse.class
Parsing MaClasse.class... Generating MaClasse.jad
```

Exemple : le fichier MaClasse.jad généré

```
// Decompiled by Jad v1.5.8g. Copyright 2001 Pavel Kouznetsov.
// Jad home page: http://www.kpdus.com/jad.html
// Decompiler options: packimports(3)
// Source File Name:   MaClasse.java

package com.jmdoudoux.test.decompile;

import java.io.PrintStream;
import java.util.*;

public class MaClasse
{

    public static void main(String args[])
    {
        MaClasse maclasse = new MaClasse("nom1", "prenom1", new Date());
        maclasse.ajouterCommande("commande 1");
        maclasse.ajouterCommande("commande 2");
        System.out.println(maclasse);
    }

    public MaClasse(String s, String s1, Date date)
    {
        commandes = new ArrayList();
        nom = s;
        prenom = s1;
        dateNaissance = date;
    }

    public void ajouterCommande(String s)
    {
        commandes.add(s);
    }

    public String toString()
    {
        StringBuilder stringbuilder = new StringBuilder("");
        stringbuilder.append("Nom : ");
        stringbuilder.append(nom);
        stringbuilder.append("\n");
        stringbuilder.append("Prenom : ");
        stringbuilder.append(prenom);
        stringbuilder.append("\n");
        stringbuilder.append("Date de naissance : ");
        stringbuilder.append(dateNaissance);
        stringbuilder.append("\n");
        stringbuilder.append("Commandes :\n");
        for(Iterator iterator = commandes.iterator();
```

```
            iterator.hasNext(); stringbuilder.append("\n"))
        {
            Object obj = iterator.next();
            stringbuilder.append("  ");
            stringbuilder.append(obj);
        }

        return stringbuilder.toString();
    }

    private String nom;
    protected String prenom;
    public Date dateNaissance;
    public List commandes;
}
```

Le code décompilé est similaire au code source original excepté :

- Le nom des variables
- L'ordre de déclaration des membres
- Les commentaires sont absents
- Le formatage est différent
- L'initialisation des attributs est déplacée dans le constructeur
- Certaines fonctionnalités de Java 5 ne sont pas décompilées à l'identique de l'original

Les fonctionnalités de Java 5 sont rarement décompilés à l'identique de l'original car ces fonctionnalités sont des raccourcis syntaxiques qui sont traités par le compilateur pour générer du code compatible avec les versions précédentes (l'annotation @override est absente, la boucle for est étendue). La décompilation restitue le code tel qu'il a été généré par le compilateur à partir du byte code : c'est notamment le cas dans l'exemple de la boucle for.

Si les informations de débogage sont incluses dans le byte code lors de la compilation, alors le résultat de la décompilation est plus proche du code source original notamment la décompilation pourra restituer le nom des variables locales et des paramètres des méthodes. Pour demander l'ajout des informations de débogage, il faut utiliser l'option -g du compilateur.

Exemple :

```
C:\java\tests>javac -g com/jmdoudoux/test/decompile/MaClasse.java
Note: com\jmdoudoux\test\decompile\MaClasse.java uses unchecked or unsafe operat
ions.
Note: Recompile with -Xlint:unchecked for details.

C:\java\tests>cd com/jmdoudoux/test/decompile

C:\java\tests\com\jmdoudoux\test\decompile>jad MaClasse.class
Parsing MaClasse.class...Overwrite MaClasse.jad [y/n/a/s] ? y
 Generating MaClasse.jad
```

Exemple : le fichier MaClass.jad généré

```
// Decompiled by Jad v1.5.8g. Copyright 2001 Pavel Kouznetsov.
// Jad home page: http://www.kpdus.com/jad.html
// Decompiler options: packimports(3)
// Source File Name:   MaClasse.java

package com.jmdoudoux.test.decompile;

import java.io.PrintStream;
import java.util.*;

public class MaClasse
{

    public static void main(String args[])
    {
        MaClasse maClasse = new MaClasse("nom1", "prenom1", new Date());
        maClasse.ajouterCommande("commande 1");
```

```
        maClasse.ajouterCommande("commande 2");
        System.out.println(maClasse);
    }

    public MaClasse(String nom, String prenom, Date dateNaissance)
    {
        commandes = new ArrayList();
        this.nom = nom;
        this.prenom = prenom;
        this.dateNaissance = dateNaissance;
    }

    public void ajouterCommande(String libelle)
    {
        commandes.add(libelle);
    }

    public String toString()
    {
        StringBuilder sb = new StringBuilder("");
        sb.append("Nom : ");
        sb.append(nom);
        sb.append("\n");
        sb.append("Prenom : ");
        sb.append(prenom);
        sb.append("\n");
        sb.append("Date de naissance : ");
        sb.append(dateNaissance);
        sb.append("\n");
        sb.append("Commandes :\n");
        for(Iterator i$ = commandes.iterator(); i$.hasNext(); sb.append("\n"))
        {
            Object commande = i$.next();
            sb.append("  ");
            sb.append(commande);
        }

        return sb.toString();
    }

    private String nom;
    protected String prenom;
    public Date dateNaissance;
    public List commandes;
}
```

46.2. Obfusquer le byte code

Pour diverses raisons, il n'est pas toujours souhaitable de proposer le code source ou de permettre son obtention grâce à une décompilation, notamment pour protéger des droits sur la propriété intellectuelle.

Il existe plusieurs outils pour obfusquer le byte code produit par le compilateur. Plusieurs outils open source ou gratuit sont utilisables pour obfusquer du byte code.

Outils	Url
ProGuard	http://proguard.sourceforge.net/
RetroGuard (Plusieurs licences dont une open source)	http://www.retrologic.com/
yGuard	http://www.yworks.com/en/products_yguard_about.htm
JavaGuard (Plus d'évolution depuis 2002)	http://sourceforge.net/projects/javaguard/
jarg (Plus d'évolution depuis 2003)	http://jarg.sourceforge.net/
JODE (Plus d'évolution depuis 2004)	http://jode.sourceforge.net/

Il existe aussi plusieurs outils commerciaux dont un des plus puissants est Klassmaster de Zelix (http://www.zelix.com/klassmaster/)

46.2.1. Le mode de fonctionnement de l'obfuscation

L'obfuscation rend parfois la décompilation impossible ou le code source produit non compilable mais plus généralement elle rend le code source issu de la décompilation très peut lisible et donc difficilement compréhensible.

L'obfuscation consiste donc à transformer le bytecode pour le rendre le moins compréhensible par un humain suite à un processus de décompilation.

Il n'existe pas de standard concernant l'obfuscation et chaque outil propose ces propres mécanismes pour obtenir un niveau de protection plus ou moins élevé. Ces mécanismes peuvent entre autre inclure :

- La suppression des informations de débogage (nom des variables, numéro de lignes, ...) : ces informations ne sont pas nécessaires à l'exécution de la classe mais sont utilisées par les débogueurs et les outils de décompilation. Si elles sont présentes le décompilateur les utilise dans le code source généré sinon il génère des noms automatiquement généralement composés d'une lettre et d'un chiffre.
- Renommage des packages, des classes, des méthodes : l'utilisation de noms explicites est important pour le développement et la maintenance du code mais ils sont inutiles pour la JVM. Ainsi si l'obfuscateur renomme ces entités avec des noms générés, cela rend la compréhension plus difficile suite à un processus de décompilation. De nombreuses classes, méthodes et variables avec le même nom rendent le code particulièrement difficile à comprendre. L'obfuscation peut aussi exploiter le polymorphisme : plusieurs méthodes ayant des noms différents avec des paramètres et une valeur de retour différents peuvent être renommées avec le même nom.
- Encodage des chaînes caractères : comme les chaînes de caractères sont stockées telle quelle dans le byte, elles sont facilement identifiables. L'obfuscation peut les encoder pour les rendre illisibles.
- Modification du flux de contrôles des traitements : l'obfuscation peut altérer le flux de contrôles des traitements notamment en faisant usage de l'instruction goto. La lecture des traitements décompilés est ainsi plus difficile à suivre car le code source devient du code « spaghetti »
- Insertion de bytecode « bogué » qui n'est jamais exécuté mais qui empêcher la décompilation. Ce bytecode exploite généralement quelques flous dans les spécifications de la JVM.

Cette transformation doit cependant garantir que le byte code modifié est toujours valide et surtout que les fonctionnalités soient toujours les mêmes.

L'outil d'obfuscation charge le fichier .class, analyse la structure et le byte code, applique les transformations et sauvegarde le résultat dans un nouveau fichier .class qui est différent de l'original mais qui doit proposer exactement les mêmes fonctionnalités.

Il est possible que l'obfuscation rende le résultat d'une décompilation non compilable grâce à l'exploitation des spécifications de Java. Une des techniques consiste à renommer des entités pour les rendre ambiguë à la compilation. Au chargement d'un fichier .class le byte code est vérifié mais certaines vérifications ne sont faites que par le compilateur et ne sont pas reproduite au chargement de la classe. Ainsi le byte code obfusqué est exécuté dans la JVM mais le résultat d'une décompilation ne se recompile pas.

La plupart des outils d'obfuscation réalise durant leur traitement une opération de shrinking qui consiste à supprimer les portions de code inutilisé : ceci permet de réduire la taille du byte code. Certain outil d'obfuscation propose aussi d'optimiser le bytecode.

L'opération d'obfuscation rend moins facilement exploitation les piles d'appels des exceptions. La plupart des outils d'obfuscation fournissent une solution pour restituer la pile d'appels telle qu'elle serait affichée avec le code non obfusqué.

46.2.2. Un exemple de mise en oeuvre avec ProGuard

ProGuard est un outil open source sous licence GPL écrit en Java qui permet d'effectuer plusieurs opérations sur une application packagée :

- Shinker qui supprime les classes, les méthodes et les champs inutilisés
- Optimisation du byte code en supprimant les instructions inutilisées
- Obfuscation en supprimant les informations de débogage et en renommant les classes, méthodes et les champs lorsque cela est possible
- Pré vérification du byte code pour Java 6 et Java ME

Pour lancer Proguard en ligne de commande, il faut exécuter la commande

java -jar proguard.jar [options ...]

Le fichier proguard.jar se trouve dans le sous répertoire lib de ProGuard.

Pour faciliter la gestion des options, il est possible de les regrouper dans un fichier de configuration. Ce fichier de configuration peut facilement être créé avec l'interface graphique fournie par ProGuard (proguardgui).

Exemple partiel du fichier config.pro :

```
-injars 'C:\java\test.jar'
-outjars 'C:\java\test.jar'

-libraryjars 'C:\Program Files\Java\jre1.6.0_05\lib\rt.jar'

# Keep - Applications. Keep all application classes, along with their 'main'
# methods.
-keepclasseswithmembers public class * {
    public static void main(java.lang.String[]);
}

# Keep - Library. Keep all public and protected classes, fields, and methods.
-keep public class * {
    public protected <fields>;
    public protected <methods>;
}
...
```

Pour lancer l'application avec un fichier de configuration, il suffit de le préciser en paramètre précédé d'un caractère @.

Exemple :

```
java -jar proguard.jar @config.pro
```

Proguard peut être utilisé via une interface graphique. Pour lancer cette interface graphique, il faut dans le répertoire lib de ProGuard la commande

Exemple :

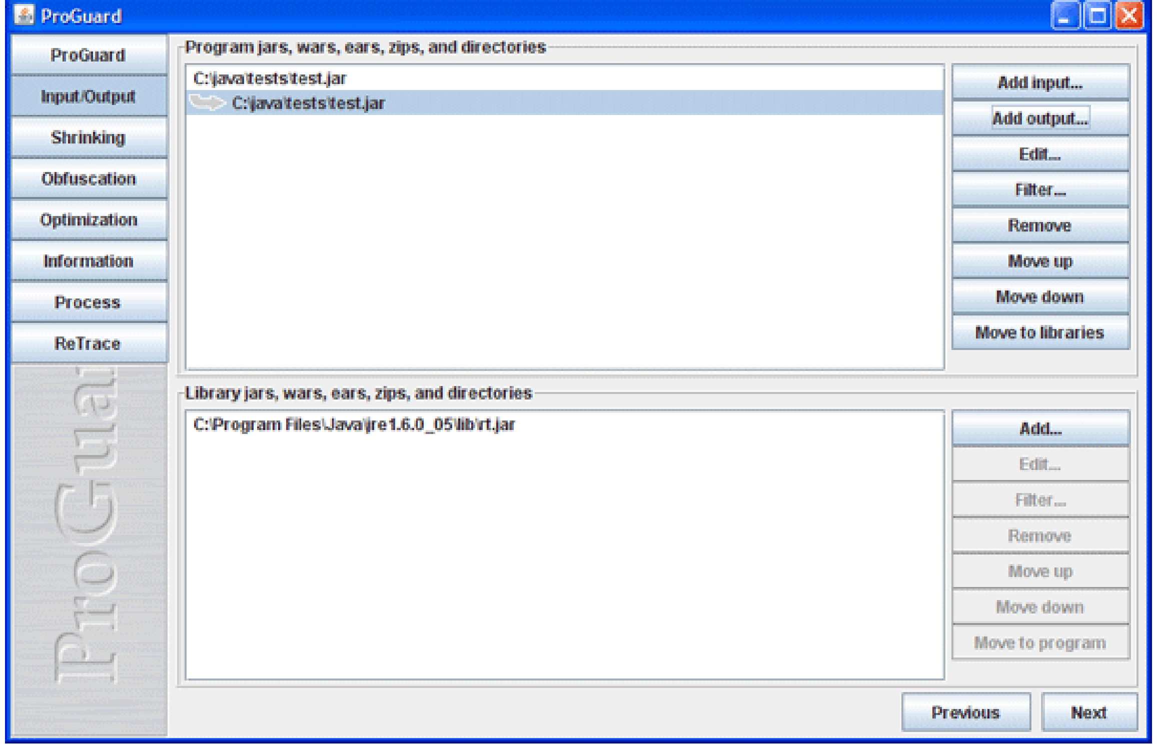
```
C:\java\proguard4.2\lib>java -jar proguardgui.jar
```

Pour utiliser ProGuard, il faut packager les fichiers .class dans une archive (de type jar, war, ...)

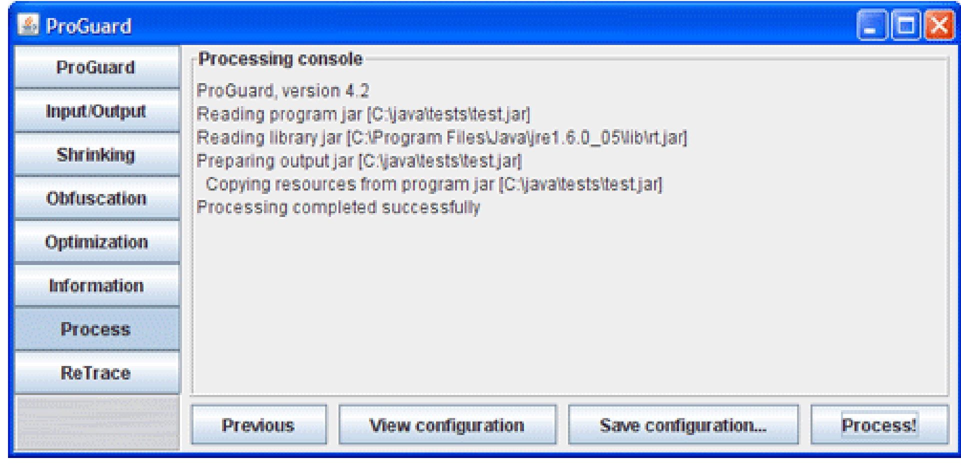
Exemple :

```
C:\java\tests>jar -cvfm test.jar manifest.mf com
manifest ajoutÚ
ajout : com/ (entrÚe = 0) (sortie = 0) (0% stockÚ)
ajout : com/jmdoudoux/ (entrÚe = 0) (sortie = 0) (0% stockÚ)
ajout : com/jmdoudoux/test/ (entrÚe = 0) (sortie = 0) (0% stockÚ)
ajout : com/jmdoudoux/test/decompile/ (entrÚe = 0) (sortie = 0) (0% stockÚ)
ajout : com/jmdoudoux/test/decompile/MaClasse.class (entrÚe = 1755) (sortie = 97
1) (44% compressÚs)
ajout : com/jmdoudoux/test/decompile/MaClasse.java (entrÚe = 1545) (sortie = 535
) (65% compressÚs)
```

Cliquez sur le bouton « Input/ouput », puis sur le bouton « Add input... » et sélectionnez le fichier test.jar précédemment créé. Cliquez sur le bouton « Add Output » et sélectionnez le fichier test.jar précédemment créé.



Cliquez sur le bouton « Process » puis sur le bouton « Process ! »



Résultat de l'exécution :

```
C:\java\tests>dir
 Volume in drive C has no label.
 Volume Serial Number is 1F23-7A9

 Directory of C:\java\tests

01/04/2008  15:31    <DIR>          .
01/04/2008  15:31    <DIR>          ..
31/03/2008  10:05    <DIR>          com
01/04/2008  15:29                51 manifest.mf
01/04/2008  15:31             2 680 test.jar
               3 File(s)          2 806 bytes
               3 Dir(s)  57 784 393 728 bytes free

C:\java\tests>dir
 Volume in drive C has no label.
 Volume Serial Number is 1F23-7A9

 Directory of C:\java\tests

01/04/2008  15:31    <DIR>          .
01/04/2008  15:31    <DIR>          ..
31/03/2008  10:05    <DIR>          com
01/04/2008  15:29                51 manifest.mf
01/04/2008  15:47             1 954 test.jar
               3 File(s)          2 080 bytes
               3 Dir(s)  57 783 062 528 bytes free
```

Pour vérifier le travail effectué par ProGuard, il faut décompiler le fichier MaClass.class obfusqué.

Exemple :

```
C:\java\tests>mkdir temp

C:\java\tests>copy test.jar temp
        1 file(s) copied.

C:\java\tests>cd temp

C:\java\tests\temp>dir
 Volume in drive C has no label.
 Volume Serial Number is 1F23-7A9

 Directory of C:\java\tests\temp

01/04/2008  15:49    <DIR>          .
01/04/2008  15:49    <DIR>          ..
01/04/2008  15:47             1 954 test.jar
               1 File(s)          1 954 bytes
               2 Dir(s)  57 783 058 432 bytes free
```

```
C:\java\tests\temp>jar -xvf test.jar
dÚcompressÚe: META-INF/MANIFEST.MF
dÚcompressÚe: com/jmdoudoux/test/decompile/MaClasse.class
dÚcompressÚe: com/jmdoudoux/test/decompile/MaClasse.java

C:\java\tests\temp>cd com/jmdoudoux/test/decompile

C:\java\tests\temp\com\jmdoudoux\test\decompile>jad MaClasse.class
Parsing MaClasse.class... Generating MaClasse.jad
```

Exemple :

```
// Decompiled by Jad v1.5.8g. Copyright 2001 Pavel Kouznetsov.
// Jad home page: http://www.kpdus.com/jad.html
// Decompiler options: packimports(3)

package com.jmdoudoux.test.decompile;

import java.io.PrintStream;
import java.util.*;

public class MaClasse
{

    public static void main(String args[])
    {
        (args = new MaClasse("nom1", "prenom1", new Date())).a("commande 1");
        args.a("commande 2");
        System.out.println(args);
    }

    private MaClasse(String s, String s1, Date date)
    {
        d = new ArrayList();
        a = s;
        b = s1;
        c = date;
    }

    private void a(String s)
    {
        d.add(s);
    }

    public final String toString()
    {
        StringBuilder stringbuilder;
        (stringbuilder = new StringBuilder("")).append("Nom : ");
        stringbuilder.append(a);
        stringbuilder.append("\n");
        stringbuilder.append("Prenom : ");
        stringbuilder.append(b);
        stringbuilder.append("\n");
        stringbuilder.append("Date de naissance : ");
        stringbuilder.append(c);
        stringbuilder.append("\n");
        stringbuilder.append("Commandes :\n");
        for(this = d.iterator(); hasNext(); stringbuilder.append("\n"))
        {
            Object obj = next();
            stringbuilder.append("  ");
            stringbuilder.append(obj);
        }

        return stringbuilder.toString();
    }

    private String a;
    private String b;
    private Date c;
    private List d;
}
```

46.2.3. Les problèmes possibles lors de l'obfuscation

L'obfuscation rend le traitement des bugs d'exploitation beaucoup plus difficile. Par exemple, un moyen efficace de comprendre et isoler un problème est d'utiliser la pile d'appels (stacktrace) d'une exception qui contient les appels des différentes méthodes. Si le nom des méthodes a été modifié, la pile d'appel devient beaucoup plus difficile à exploiter vis-à-vis du code source. La pile d'appel peut aussi être plus efficace si elle exploite les informations de débogage. Hors généralement, ces informations sont supprimées lors de l'obfusaction.

L'obfuscation doit garantir que le bytecode obfusqué propose les mêmes fonctionnalités que le bytecode initial. Cependant les transformations réalisées par les outils d'obfuscation peuvent parfois avoir des effets de bords importants notamment avec certaines technologies de Java :

- L'introspection repose sur l'accès dynamique de certaines entités grâce à leur nom. La modification de ces noms entraine inévitablement des problèmes lors de l'utilisation de l'introspection à l'exécution.
- Le chargement dynamique de classe via les méthodes Class.forName() ou ClassLoader.loadClass() utilise le nom de la classe pour s'exécuter. Si la classe est renommée, cela lèvera une exception de type ClassNotFoundException à l'exécution. Ceci est d'autant plus vrai si le nom de la classe n'est pas fourni en dur mais contenu dans une variable dont la valeur est déterminée dynamiquement (par exemple à la lecture d'un fichier de configuration)
- La sérialisation d'un objet inclus des informations sur la classe. Si la classe est modifiée ou son numéro de version SerialIUD sont modifié cela empêche la désérialisation. Ainsi il n'est pas possible de sérialiser un objet et de le désérialiser avec sa version obfusquée.
- Certaines API nécessitent le respect de convention de nommage strictes de certaines méthodes (exemple avec le EJB avant leur version 3.0 : les méthodes ejbCreate() et ejbRemove())

46.2.4. L'utilisation d'un ClassLoader dédié

Pour rendre la décompilation plus difficile, il est possible d'encoder les fichiers .class avec un algorithme de cryptage et d'utiliser un ClassLoader dédié qui va décrypter les fichiers .class avant de les charger en mémoire.

Ainsi, les fichiers .class ne peuvent plus être décompilés puisque le byte code est illisible. Cependant cette technique est loin d'être infaillible car il suffit de décompiler le ClassLoader pour obtenir l'algorithme de décryptage et de l'utiliser pour décrypter les fichiers .class qui pourront ainsi être décompilés.

47. Terracotta

Le clustering est un terme qui désigne une solution technique dont le but est d'améliorer la disponibilité (fail-over) et/ou la montée en charge (load-balancing) d'une application. Concrètement cela se traduit par un groupement de serveurs qui sont vus comme un seul serveur logique. Cette redondance peut être utilisée pour :

- le fail-over : l'exécution des traitements est réalisée sur les serveurs actifs évitant ainsi de ne pas avoir de réponse si l'unique serveur est indisponible
- le load-balancing : la charge est répartie sur chacun des serveurs actifs

Toutes les solutions de clustering sont propriétaires puisqu'aucune des plateformes Java (même Java EE) ne proposent de spécifications relatives au clustering.

Pour permettre une meilleure montée en charge et un meilleur taux de disponibilité, les applications Java sont de plus en plus réparties sur différentes JVM.

Terracotta est une solution open source puissante de clustering au niveau de la JVM qui permet de facilement mettre en cluster une application dans plusieurs JVM. Ceci permet à une application d'être exécutée dans plusieurs JVM, Terracotta prenant en charge de façon transparente les interactions entre ces JVM pour faire en sorte que l'application s'exécute comme dans une seule JVM.

Terracotta fournit un environnement d'exécution qui facilite grandement la mise en cluster d'une application Java en proposant la mise en cluster au niveau de la JVM plutôt que la mise en cluster de l'application.

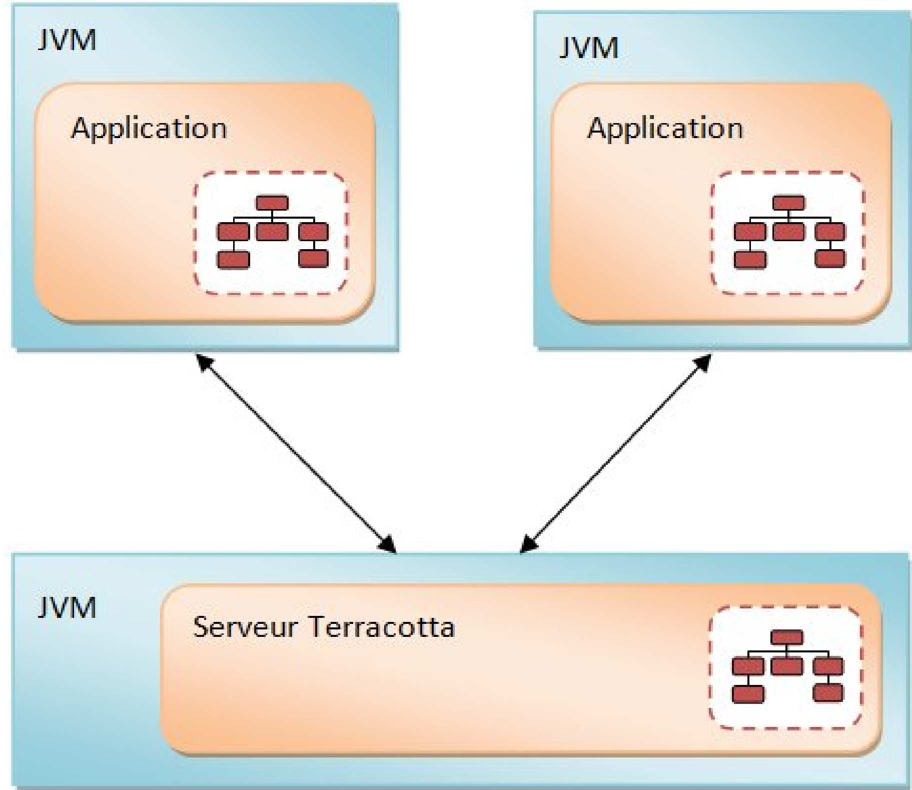
Le coeur de Terracotta est composé de deux parties :

- un serveur qui stocke les objets et coordonne les éléments du cluster
- une librairie qui permet la manipulation du byte code pour ajouter des traitements gérant interactions avec la JVM et le serveur

Terracotta agit directement sur les JVM pour maintenir l'état des objets gérés dans ces différentes JVM notamment en permettant entre autre :

- le partage d'objets
- le maintient des références
- la gestion des accès concurrents
- la synchronisation des threads (méthode wait(), notify(), notifyAll() de la classe Object)
- la gestion du ramasse miettes
- ...

L'état des objets gérés est conservé dans le serveur (ou un ensemble de serveurs) qui est une application Java donc exécutée dans sa propre JVM.



Terracotta permet à une application exécutée dans plusieurs JVM d'accéder à des objets partagés dans une mémoire virtuelle.

Terracotta permet le partage de graphes d'objets entre plusieurs JVM sans avoir recourt à une API dédiée : pour cela, Terracotta manipule dynamiquement le byte code de l'application ce qui évite d'avoir à mettre en oeuvre une API particulière dans le code de l'application. Il n'est donc pas utile de mettre en oeuvre des API dédiées à l'échange de données comme des sockets, RMI, JMS, des services web, ...

L'accès aux données se fait de façon transparente : Terracotta s'occupe de stocker les données sur le serveur, de la copier localement lors de leur utilisation et de maintenir leur état sur le serveur et sur les clients.

Côté client, un classloader particulier instrumente les objets dans la JVM pour assurer le dialogue avec le serveur. Ainsi, si l'application est prévue pour fonctionner en multi-thread dans une JVM, elle pourra bénéficier de sa mise en cluster grâce à Terracotta sans altération de son code source.

Les échanges entre les JVM et le serveur sont particulièrement optimisés : ils ne reposent pas sur le transfert complet des objets sérialisés mais utilisent un mécanisme propre à Terracotta qui réduit les échanges au minimum.

Le fichier de configuration tc-config.xml permet de configurer la mise en oeuvre de Terracotta.

Terracotta est un projet open source pour lequel une version commerciale permet d'avoir du support et des fonctionnalités avancées.

Le site web officiel est à l'url : http://www.terracotta.org/

La version de Terracotta utilisée dans ce chapitre est la 3.2.

47.1. Présentation de Terrocatta

Terracotta permet à une application d'être déployée sur plusieurs JVM et de s'exécuter comme si elle l'était sur une seule. Terracotta permet d'appliquer le modèle de gestion de la mémoire d'une JVM dans plusieurs JVM qui peuvent être sur différentes machines.

En pratique, Terracotta permet de voir un cluster de JVM comme une unique JVM pour le développeur : il est par exemple possible d'utiliser un singleton dans le cluster sans code supplémentaire.

Terracotta permet le partage d'objets et la coordination de threads entre différentes JVM.

La gestion des accès concurrents est assurée simplement en utilisant les mécanismes fournis par Java (moniteur via le mot clé synchronized, utilisation de la bibliothèque java.util.concurrent, ...).

Terracotta prend aussi en charge des fonctionnalités Java de base au travers du cluster comme la gestion des accès concurrents (synchronized), la gestion de la mémoire (ramasse miette), la gestion des threads (wait(), notify(), ...). Terracotta prend en charge ces mécanismes de façon distribuée sous réserve qu'ils soient mis en oeuvre dans l'application par le développeur.

Les mécanismes de sérialisation, très couteux, ne sont pas utilisés et les modifications ne sont pas broadcastées à tous les noeuds du cluster mais simplement à ceux qui en ont besoin : ceci permet de maximiser les performances des échanges réseaux réalisés par Terracotta.

Les objets partagés dans le cluster sont nommés Distributed Shared Objects (DSO). Ces objets sont vus comme des objets standards dans le heap de chaque client mais c'est Terracotta qui se charge de la gestion de ces objets de façon transparente grâce à l'instrumentation des objets gérés et de ceux qui les utilisent.

Terracotta met en oeuvre un ramasse miette distribué (DGC : Distributed Garbage Collector) qui s'assure avant de récupérer la mémoire d'un objet que celui n'a plus de références sur le serveur mais aussi sur chacun des clients.

47.1.1. Le mode de fonctionnement

Les solutions de mise en oeuvre de cluster d'application Java repose généralement sur la sérialisation des objets modifiés pour les répliquer dans les différentes JVM. Ce type de solution est coûteux du fait même de l'utilisation de la sérialisation :

- coût de la sérialisation
- échange de l'intégralité de l'objet même si un seul champ a été modifié

Terracotta propose une solution alternative qui réduit ces coûts en n'échangeant que les modifications faites sur un objet. Il utilise son propre mécanisme natif pour l'échange des données : il n'est donc pas utile que les objets gérés soient sérialisables. L'état d'un objet est stocké sur un serveur et les clients ne reçoivent les modifications que lorsque l'objet est utilisé en local. Terracotta se charge alors de rafraichir l'objet à partir du serveur de façon transparente. Les échanges réseaux sont ainsi réduits.

Terracotta met en oeuvre le principe de Network Attached Memory (NAM). Ceci permet d'avoir une sorte de super JVM qui coordonne les JVM clientes du cluster. Comme certains objets et les threads sont partagés et synchronisés, les clients du cluster sont vus comme s'il ne s'agissait que d'une seule JVM.

La définition des objets gérés par Terracotta se fait dans un fichier de configuration au format XML : cette définition peut se faire de façon très fine et évite d'avoir à partager tous les objets de chaque JVM cliente.

Le byte code de certaines classes de l'application est enrichi au chargement des classes : ceci permet une instrumentation des classes grâce à un classloader dédié.

La bibliothèque Terracotta analyse chaque classe chargée définie dans le fichier de configuration et injecte du byte code au besoin pour permettre de dialoguer avec le serveur et effectuer les traitements que le serveur impose (mise à jour de l'état d'un ou plusieurs objets, pose ou libération de verrous sur les moniteurs, synchronisation des threads, ...).

L'instrumentation du byte code permet donc entre autre de capturer les modifications faites sur un objet et de les envoyer au serveur. Ces modifications ne seront envoyées aux autres noeuds que lorsque ceux ci auront besoin d'accéder à l'objet. Ainsi tous les noeuds ont accès aux objets mais ceux ci ne sont mis à jour dans la JVM local qu'aux besoins.

Terracotta assure la cohérence d'une donnée gérée au travers du cluster.

Les objets gérés sont stockés dans le serveur Terracotta. Terracotta définit une racine pour un graphe d'objets qui sont partagés. Cette racine est identifiée par un champ nommé root dans le fichier de configuration.

Lorsque d'un objet racine est instancié, lui même et toutes ces dépendances sont partagées par Terracotta. Toutes les données de ces objets sont stockées sur le serveur Terracotta.

Lors d'une modification sur l'état d'un objet géré, celle-ci est automatiquement répercutée au besoin dans les autres noeuds du cluster par le serveur Terracotta.

Terracotta utilise la notion de transaction pour gérer les accès concurrents aux objets gérés dans le cluster. Cette gestion repose sur les mêmes mécanismes que ceux utilisés dans une JVM mais de façon distribuée entre les noeuds du cluster : cette distribution se fait par l'intermédiaire du serveur.

Ainsi les méthodes synchronized, les blocs synchronized et les méthodes déclarées locked dans le fichier de configuration de Terracotta sont utilisés comme délimiteurs pour ces transactions.

A la fin de la transaction, les modifications sont envoyées au serveur pour les diffuser aux autres noeuds du cluster.

Un objet géré par le cluster vie dans la JVM d'un noeud de la même façon qu'un objet non géré sauf que son état est synchronisé par le serveur :

- lorsque des modifications sont faites en locales, l'objet est directement modifié et les modifications sont envoyées au serveur
- lorsque des modifications sont faites dans un noeud distant, celles ci sont fournies à la JVM par le serveur et les modifications sont appliquées à l'objet local

Chaque noeud possède sa propre instance d'un objet géré par le cluster dont l'état est synchronisé grâce au serveur du cluster.

Aucune API liée à Terracotta n'est à mettre en oeuvre dans le code pour faire fonctionner le cluster. Cependant, certaines modifications dans le code sont parfois nécessaires notamment pour améliorer les performances : ces modifications ne sont pas directement liée à Terracotta mais à la bonne mise en oeuvre d'une programmation cocurrente. La transparence de Terracotta est possible grâce à l'instrumentation du byte de code des classes à leur chargement dans la JVM.

Pour le développeur, la seule préoccupation est de s'assurer de la bonne gestion des accès concurrents sur les objets gérés.

47.1.2. La gestion des objets par le cluster

Terracotta utilise le concept de virtual heap qui permet de gérer dans le cluster de grande quantité de données qui peuvent être largement supérieure à la mémoire disponible sur les clients. Ceci est possible car les objets gérés par le cluster ne sont fournis aux clients qu'au moment où ceux ci en ont besoin.

L'instrumentation du byte code côté client permet au moment de l'accès à un champ de demander sa valeur au serveur et de l'instancier dans la mémoire locale si celui ci n'existe pas encore en local. Il est possible que ces références locales soient supprimées localement par le ramasse miette de la JVM. Si cette référence est de nouveau requise en local, elle sera de nouveau obtenue du serveur.

Terracotta garantit qu'une instance d'un objet géré par le cluster sera unique pour un classloader d'une JVM. Toutes les modifications sont répliquées sur les clients du cluster pour la même instance du classloader. Pour faciliter ce travail, Terracotta génère et attribut un identifiant unique qui lui est propre aux objets gérés dans le cluster.

Lors du chargement d'une classe, Terracotta utilise un classloader particulier pour enrichir le byte code des classes à instrumenter définies dans le fichier de configuration. Les traitements ajoutés permettent à Terracotta de synchroniser l'état des objets gérés dans le cluster dans les différentes JVM qui le compose.

Les instances des objets gérés par le cluster sont créées grâce aux traitements ajoutés à la classe lors de son instrumentation :

- Une vérification de l'existence de l'instance sur le serveur est faite
- Si elle existe alors elle est créée localement à l'image de l'état de l'objet sur le serveur

- Si elle n'existe pas alors elle est créée localement et elle est répliquée sur le serveur

Les classes qui accèdent à des objets gérés par le cluster doivent aussi être instrumentées même si elles ne sont pas elle-même gérées par le cluster.

Les dépendances d'un objet géré par le cluster sont automatiquement gérées par le cluster : ceci permet aux graphes d'objets des objets racines d'être gérés automatiquement par le cluster assurant ainsi de maintenir la cohérence de l'état de l'objet racine au travers du cluster.

Un objet peut être géré de deux façons dans le cluster Terracotta :

- Physically Managed : c'est la façon la plus courante dans laquelle les modifications faites dans chaque champ sont envoyées au serveur qui les reporte directement dans l'objet à chaque noeud qui possède une instance locale de l'objet.
- Logically Managed : pour certains objets particuliers (généralement proche de la JVM comme les classes qui utilisent un hashcode) les modifications sont propagées par enregistrement de l'invocation des méthodes avec leur paramètre et leur invocation distribuée sur les noeuds du cluster.

Certaines classes ne peuvent pas être gérées dans le cluster comme par exemple la classe Thread. Il en va de même pour les classes qui héritent de ces classes même si elles sont instrumentées. Si une classe non gérable est incluse dans le graphe d'objets gérés par le cluster alors une exception de type TCNonPortableObjectException est levée.

Il est possible de définir certains champs d'un objet gérés par le cluster comme transient. Ces champs ne seront alors pas gérés par le cluster.

Par défaut, les champs marqués avec le modificateur transient ne sont pas ignorés par Terracotta mais il possible de demander qu'ils soient ignorés via le fichier de configuration. N'importe quel champ peut aussi être ignoré grâce à une définition particulière dans le fichier de configuration.

Terracotta propose de définir des traitements qui seront exécutés à l'instanciation de la classe qui permet une initialisation correcte des champs transient.

47.1.3. Les avantages de Terracotta

L'élégance de cette solution est de ne pas être intrusive dans le code de l'application mise en cluster :

- aucune API liée à Terracotta n'est à utiliser
- les objets gérés sont de simple POJO : aucune interface n'est à implémenter, pas même Serializable

Le grand intérêt est de pouvoir utiliser une application de façon distribuée via plusieurs instances dans des JVM dédiées sans avoir à modifier le code de l'application. Terracotta permet une mise en oeuvre d'un cluster de façon transparente pour le développeur : il n'y a pas de code intrusif à rajouter pour faire fonctionner l'application en cluster, sous réserve que le code soit déjà prévu pour fonctionner dans un mode multi-threads.

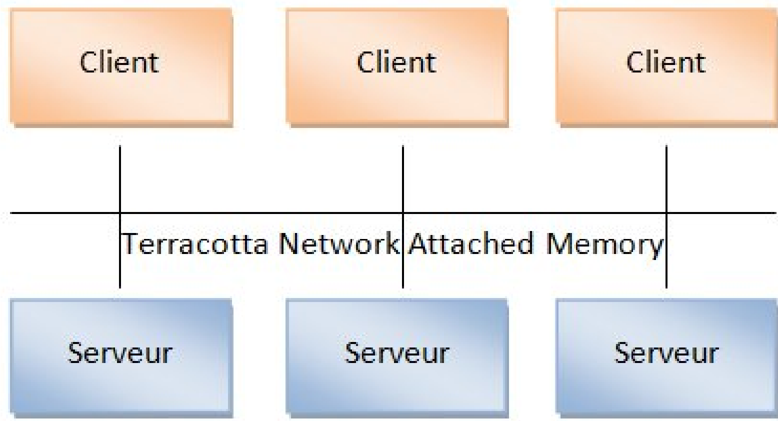
47.1.4. L'architecture d'un cluster Terracotta

Le projet Terracotta met en oeuvre le clustering au niveau de la JVM en utilisant des mécanismes de partage de mémoire au travers du réseau (NAM : Network Attached Memory) qui permettent de partager des instances d'objets entre plusieurs JVM.

L'architecture d'un cluster Terracotta est composée de deux types d'éléments : des noeuds clients et un ou plusieurs serveurs :

- les noeuds clients : les JVM qui exécutent l'application mise en cluster sont appelées des clients Terracotta ou des noeuds du cluster Terracotta
- un ou plusieurs serveurs Terracotta qui agissent comme un gestionnaire d'objets et de threads partagés entre tous les clients du cluster comme si il ne s'agissait que d'une seule JVM. Le serveur Terracotta stocke les objets

gérés, synchronise l'état de ces objets dans les différents noeuds et coordonne les threads entre les JVM.



Un serveur Terracotta assure plusieurs fonctionnalités :

- Gestion des objets gérés par le cluster et synchroniser leur état avec les clients
- Gérer les verrous distribués et coordonner les threads des différents clients
- Eventuellement persistance des objets sur disque pour assurer une persistance ou un stockage volatile (stockage temporaire sur disque des objets suite à un manque de place mémoire sur le serveur)

Le serveur gère les verrous posés ou demandés pour les différents threads des clients. Il coordonne aussi les notifications sur les threads concernés des clients lors de l'utilisation des méthodes notify() et notifyAll() sur des objets gérés par le cluster.

Le serveur gère les objets dont il a la charge : ces objets sont instanciés sur un client qui en informe le serveur pour stockage et diffusion aux autres clients selon leurs besoins. Le serveur fournit aussi l'état des objets lors des sollicitations des clients pour qu'il puisse créer une instance en local.

Il est possible de mettre en place plusieurs serveurs Terracotta en mode fail over. Un de ces serveurs est le serveur actif, les autres sont passifs (un des serveurs passifs prend le relai en devenant actif en cas d'arrêt du serveur actif). Ce sont les clients qui vont essayer de se connecter au serveur actif.

Le serveur Terracotta peut sauvegarder les données des objets partagés sur disque pour ne pas les perdre en cas d'arrêt du serveur.

47.2. Les concepts utilisés

Terracotta met en oeuvre plusieurs concepts :

- Un objet racine (root) : défini dans le fichier de configuration, il permet à Terracotta de savoir qu'un objet et ses dépendances sont gérés dans le cluster
- les transactions : elles permettent de garantir la cohérence des modifications faites sur des objets
- les verrous : ils permettent de gérer les accès concurrents faits sur les objets gérés par le cluster
- l'instrumentation des classes : au chargement de chaque classe précisée dans le fichier de configuration, une instrumentation de ces classes est réalisée pour permettre une communication avec le serveur.
- le ramasse miette distribué (Distributed Garbage Collector) : libère la mémoire des objets du serveur qui n'ont plus de référence ni sur le serveur ni sur les clients

47.2.1. Les racines (root)

Tous les objets d'un noeud d'un cluster Terracotta ne sont pas gérés par le cluster mais uniquement ceux qui sont déclarés comme étant la racine (root) d'un graphe d'objets gérés par le cluster.

Une racine permet d'identifier un objet comme devant être géré dans le cluster. La définition d'une racine se fait de manière déclarative dans le fichier de configuration. Une racine (root) constitue le sommet d'un graphe d'objets qui seront gérés et partagés par le cluster Terracotta. Le graphe est constitué de la racine et des objets qui sont accessibles depuis cette instance.

Une racine est un champ d'une classe dont le nom pleinement qualifié est défini dans le fichier de configuration.

Lorsqu'une racine est instanciée pour la première fois par un client du cluster, l'instance et ses éventuelles dépendances sont créées sur le serveur Terracotta.

Une fois qu'une instance d'une racine est créée sur le serveur, celle ci ne peut plus être réaffectée : ceci n'empêche pas de modifier les autres instances du graphe d'objets de l'instance.

Une racine peut être une variable de type littéral :

- types primitifs : byte, short, int, long, char, float, double, boolean,
- leur Wrapper : Byte, Short, Integer, Long, Character, Float, Double, Boolean
- certains objets : String, Class, BigInteger, BigDecimal
- les énumérations

La valeur d'un objet racine de type littéral peut évoluer durant le cycle de vie du cluster. Les verrous sur un littéral sont posés par Terracotta sur leurs valeurs et non sur leur référence.

La classe qui encapsule la racine doit obligatoire être instrumentée par Terracotta.

Des dépendances d'un objet géré par le cluster peuvent être configurées pour ne pas être gérées par le cluster.

47.2.2. Les transactions

Une transaction est un ensemble de modifications faites de façon atomique sur un ou plusieurs objets gérés par le cluster afin de garantir la cohérence de leur état.

La portée d'une transaction est définie grâce à la pose et la libération d'un verrou. Lorsqu'un verrou est posé, Terracotta débute une transaction qui va enregistrer les modifications faites dans les objets gérés. Lorsque le verrou est libéré, la transaction est validée en reportant les modifications sur le serveur.

Chaque changement sur un objet géré par le cluster doit se faire dans une transaction : le thread qui veut faire la modification doit obligatoirement poser un verrou avant de changer l'état d'un objet géré par le cluster sinon une exception de type UnlockedSharedObjectException est levée.

Il faut donc obligatoirement :

- que les modifications d'un objet géré se fassent dans une transaction
- que l'objet soit géré par le cluster avant d'être modifié dans une transaction

Aucune transaction n'est définie pour une méthode :

- qui ne soit pas définie dans la partie lock du fichier de configuration
- qui n'est pas synchronized ou qui n'a pas l'attribut auto-synchronized à true

Important : il est impératif d'instrumenter toutes les classes qui peuvent modifier l'état d'un objet géré par le cluster. Dans le cas contraire, Terracotta ne verra pas les modifications et l'objet pourra être dans un état inconsistant.

47.2.3. Les verrous (lock)

Les modifications faites sur des objets gérés par le cluster doivent l'être dans le cadre d'une transaction. Une transaction enregistre les modifications sur les objets et les données primitives des objets. A la fin de la transaction, les modifications

faites sur les objets sont envoyées au serveur.

Les verrous ont deux utilités essentielles dans le fonctionnement de Terracotta :

- Permettre de coordonner les accès aux sections critiques de code entre les threads des JVM
- Permettre de déterminer le début et la fin d'une transaction

Les verrous sont assez similaires au rôle du mot clé synchronized. D'ailleurs la définition d'un verrou peut se faire via le mot clé synchronized et/ou par définition dans le fichier de configuration.

Les verrous sont définis dans le fichier de configuration grâce à des expressions régulières qui définissent une méthode ou un ensemble de méthodes.

Il est impératif que les classes des méthodes sur lesquelles des verrous sont définis soient instrumentées.

Les verrous sont définis dans le fichier de configuration avec les éléments <autolock> parent de l'élément <locks>.

Ceci permet à Terracotta d'ajouter aux méthodes définies qui possèdent une synchronisation sur leur classe le code nécessaire à la pose de verrous au travers du cluster.

Exemple :

```
<locks>
  <autolock auto-synchronized="false">
    <method-expression>* com.jmdoudoux.test.MaClasse.*(..)</method-expression>
    <lock-level>write</lock-level>
  </autolock>
</locks>
```

La ou les méthodes qui seront verrouillées sont identifiées avec une expression dont la syntaxe est celle d'AspectWerkz.

Le type de verrou est précisé dans le fichier de configuration avec un tag <lock-level>.

Les verrous de Terracotta peuvent être posés selon plusieurs niveaux :

- Read : permet de s'assurer que les données partagées lues sont fraiches. Plusieurs threads peuvent poser un verrou mais uniquement pour de la lecture : aucune modification n'est possible sur les objets gérés par le cluster dans ce type de verrou. Aucun thread du cluster ne peut obtenir un verrou de type write si un verrou de type read est posé. Si un thread modifit un objet géré par le cluster dans une transaction avec un verrou de type read alors une exception de type UnlockedSharedObjectException est levée. Une exception est aussi levée si un thread pose un verrou de type write alors qu'un verrou de type read est déjà posé
- Synchronous write : un seul thread peut accéder à la portion de code. Le verrou sera conservé jusqu'à ce que toutes les modifications ont été reportées sur le serveur et acquittées par le serveur
- Write : agit comme les verrous Java. Un seul thread dans tout le cluster peut poser le verrou et accéder à la fois
- Concurrent : verrou utilisé en interne par Terracotta

Le type et la portée d'un verrou peuvent avoir de gros impacts sur les performances de l'application exécutée dans le cluster.

Il n'est pas toujours possible de modifier le code existant pour ajouter des portions de code synchronized. Terracotta permet de les ajouter lors de l'instrumentation des classes grâce à deux fonctionnalités :

- auto-synchronized : permet d'ajouter le mot clé synchronized à la méthode concerné. Le verrou ainsi défini peut cependant être de type read ou write selon la définition réalisée dans le fichier de configuration
- named lock : permet de définir un verrou global qui va s'appliquer à toutes les instances de la classe au travers du cluster

Il peut être tentant de mettre auto-synchronized à toutes les méthodes mais l'effet sur les performances serait similaire à mettre le mot clé synchronized sur toutes les méthodes : les performances sont alors catastrophiques car il y a énormément de contention. Les verrous doivent donc être posés judicieusement et de préférence dans le code.

La pose de verrous est ajoutée par Terracotta sur les méthodes synchronized et leurs portions de code synchronized qui sont identifiés dans un autolock.

Une méthode définie dans un autolock qui n'a pas de synchronized n'a aucun effet. Dans ce cas, il est possible d'utiliser l'attribut auto-synchronized pour que Terracotta ajout dynamiquement le mot clé synchronized à la méthode.

La mise en oeuvre des autolock de Terracotta se fait lors de l'instrumentation du byte code (MONITORENTER et MONITOREXIT) en ajoutant du byte code pour poser et lever un verrou au niveau du cluster.

Le verrou doit donc se faire sur un objet géré par le cluster car l'acquisition du verrou se fait en utilisant l'identifiant unique attribué par Terracotta. Pour une méthode synchronized qui n'est pas définit dans la partie autolock, le verrou est posé uniquement dans la machine virtuelle locale.

Terracotta propose, en plus des autolock, des named locks qui permettent de définir des verrous identifiés par un nom. Il est préférable d'utiliser des autolock plutôt que des name locks lorsque cela est possible car ces derniers peuvent avoir de gros impacts sur les performances de l'application.

Exemple :

```
<named-lock>
   <lock-name>MonVerrou</lock-name>
   <method-expression>* com.jmdoudoux.test.MaClasse.*(..)</method-expression>
   <lock-level>read</lock-level>
</named-lock>
```

47.2.4. L'instrumentation des classes

Les traitements relatifs au clustering sont ajoutés par instrumentation du byte code de l'application : ce code est injecté au chargement de la classe par un classloader dédié. Il permet notamment de poser les verrous et échanger l'état des objets gérés par le cluster avec le serveur.

Les classes à instrumenter sont définies dans le fichier de configuration. Il est inutile d'instrumenter toutes les classes mais uniquement celle qui doivent être gérées par le cluster ou qui manipule des objets gérés par le cluster. C'est d'autant plus important de limiter le nombre de classes à instrumenter que cette instrumentation est coûteuse au chargement de la classe et à l'exécution du fait des échanges réseau avec le serveur.

La définition des classes à instrumenter dans le fichier de configuration est indépendante de la définition des racines (roots) et des verrous (locks) : la définition de l'un ou l'autre n'implique pas une instrumentation explicite.

Terracotta doit instrumenter certaines classes par le classloader de boot. Ces classes ne peuvent pas être instrumentées dynamiquement puisque chargées avant Terracotta.

Celles ci doivent donc être pré-instrumentées et placée dans un fichier jar particulier qui sera ajouté dans le classpath de boot. Ce fichier jar est nommé "boot jar" par Terracotta.

Terracotta propose un outil dédié pour générer ce boot jar.

Une classe qui est chargée par le classloader de boot ne peut pas être gérée par le cluster : cette classe doit être instrumentée dans le boot jar.

La bibliothèque boot jar est précisée au lancement de chaque JVM cliente avec l'option :

-Xbootclasspath/p:chemin_du_fichier_jar_de_boot_terracotta.jar

47.2.5. L'invocation distribuée de méthodes

L'invocation distribuées de méthodes (DMI : Distributed Method Invocation) permet l'invocation d'une méthode d'un objet géré par le cluster dans tous les noeuds du cluster.

L'objet sur lequel la méthode sera invoquée doit être instrumenté et être géré par le cluster.

47.2.6. Le ramasse miette distribué

Un serveur Terracotta dispose d'un ramasse miette distribué (DGC Distributed Garbage Collector) dont le rôle est de purger les objets gérés par le cluster qui ne sont plus référencés ni dans le serveur ni dans aucun client.

Malgré son nom, ce n'est pas un processus distribué : il supprime uniquement des objets côté serveur (en mémoire et éventuellement dans le système de persistance des données du cluster) après s'être assuré qu'il n'est plus référencé par aucun objet gérés sur le serveur et qu'aucun client n'en possède encore une référence.

Un objet est géré par le cluster de son instanciation jusqu'à sa libération par le ramasse miette distribué (DGC Distributed Garbage Collector) de Terracotta.

Les objets racines ne sont pas traités par le DGC.

Il y a plusieurs façons d'exécuter le DGC :

- périodiquement selon le paramétrage dans le fichier de configuration
- à la demande via JMX
- à la demande via la console d'administration de Terracotta
- la demande en exécutant le script run-dgc

L'activité du DGC est journalisée dans le fichier de log du serveur. Cette activité peut aussi être surveillée en utilisant la console d'administration de Terracotta.

47.3. La mise en oeuvre des fonctionnalités

Cette section va fournir quelques informations pour mettre en oeuvre Terracotta.

47.3.1. L'installation

Terracotta peut être téléchargé gratuitement sur le site terracotta.org après un enregistrement obligatoire.

Il faut télécharger le fichier terracotta-3.2.0-installer.jar et l'exécuter :

C:\java>java -jar terracotta-3.2.0-installer.jar

Un assistant guide l'installation qui se fait par défaut dans le répertoire C:\Program Files\terracotta\terracotta-3.2.0

Pour faciliter son utilisation, il est possible d'ajouter le sous répertoire bin issu de l'installation à la variable système PATH.

Le répertoire d'installation contient plusieurs sous répertoires :

Nom	Contenu
bin	les scripts de commande
config-examples	les exemples de fichier de configuration

distributed-cache	ehcache
docs	la javadoc et un lien vers la doc en ligne
hibernate	
icons	
lib	les bibliothèques de Terracotta et de ses dépendances
modules	les modules permettant l'intégration de différentes technologies
quartz-1.7.0	l'api de schreduling Quartz
samples	des exemples
schema	le schéma du fichier de configuration et sa documentation
tools	des outils notamment le sessions-configurator
vendors	des outils tiers préconfigurés avec Terracotta

47.3.2. Les modules d'intégration

Pour faciliter la mise oeuvre sans une connaissance approfondie de certains outils ou bibliothèques, Terracotta propose des modules d'intégration (TIM : Terracotta Integration Module).

Ces modules proposent une configuration out of the box :

- pour certains produits : Spring, Tomcat, GlassFish,
- pour certaines API : collections, ...
- pour des frameworks courants : EHCache, Hibernate, Spring, Struts, ...

La commande tim-get permet de gérer les modules installés avec Terracotta.

La syntaxe de cette commande est de la forme :

```
tim-get.bat [command] [arguments] {options}
```

Elle possède en premier argument une commande en fonction de l'action à réaliser :

Option	Rôle
list	obtenir une liste des TIM utilisables
install	installer un module
install-for	installer les modules précisés dans le fichier de configuration
info	afficher des informations détaillées sur un module
update	installer la dernière version d'un module
upgrade	mettre à jour le fichier de configuration et installer les dernières versions des modules
help	afficher une aide sur les commandes

L'option -h ou --help de chaque commande permet d'afficher une aide sur les options de la commande.

Cette commande requiert un accès à internet.

Certaines propriétés de la commande peuvent être modifiées dans le fichier tim-get.properties du sous répertoire lib/resources d'installation de Terracotta. C'est notamment le cas si l'accès à internet passe par un proxy.

Les modules contiennent un fichier terracotta.xml qui contient un fragment de la configuration qui sera fusionné avec le

fichier de configuration de Terracotta.

47.3.3. Les scripts de commande

Terracotta propose un ensemble de scripts permettant de mettre en oeuvre le cluster.

Ces scripts sont livrés pour Unix (*.sh) ou windows (*.bat).

<table>
<tr><td>Script</td><td>Rôle</td></tr>
<tr><td>archive-tool</td><td>collecter des informations sur l'environnement pour les fournir au support de Terracotta</td></tr>
<tr><td>boot-jar-path</td><td>utiliser l'outil dso-env pour déterminer le chemin de la JVM. Cet outil ne devrait pas être utilisé directement</td></tr>
<tr><td>dev-console</td><td>lancer la console du développeur qui est outil graphique pour monitorer et piloter le cluster</td></tr>
<tr><td>dso-env</td><td>permet de configurer un environnement client en valorisant une variable d'environnement système TC_JAVA_OPTS à partir des informations fournies par les variables d'environnement système JAVA_HOME, TC_INSTALL_DIR et TC_CONFIG_PATH

Microsoft Windows

<code>set TC_INSTALL_DIR="C:\Program Files\terracotta\terracotta-3.2.0"
set TC_CONFIG_PATH="localhost:9510"
call "%TC_INSTALL_DIR%\bin\dso-env.bat" -q
set JAVA_OPTS=%TC_JAVA_OPTS% %JAVA_OPTS%
call "%JAVA_HOME%\bin\java" %JAVA_OPTS% ...</code>

UNIX/Linux (bash)

<code>TC_INSTALL_DIR="/usr/local/terracotta-3.2.0"
export TC_INSTALL_DIR
TC_CONFIG_PATH="localhost:9510"
export TC_CONFIG_PATH
. ${TC_INSTALL_DIR}/bin/dso-env.sh -q
JAVA_OPTS="${JAVA_OPTS} ${TC_JAVA_OPTS}"
${JAVA_HOME}/bin/java ${JAVA_OPTS} ...</code></td></tr>
<tr><td>dso-java</td><td>permet de lancer un environnement d'exécution pour un client du cluster</td></tr>
<tr><td>make-boot-jar</td><td>permet de créer si nécessaire le boot jar qui devra être ajouté au classpath de boot. Le boot jar est spécifique à une plateforme, une JVM et la version de cette JVM</td></tr>
<tr><td>run-dgc</td><td>demander l'exécution du ramasse miette distribué</td></tr>
<tr><td>scan-boot-jar</td><td>permet de vérifier la cohérence entre le boot jar et le fichier de configuration</td></tr>
<tr><td>server-stat</td><td>(server status tool) permet de vérifier le statut courant du cluster</td></tr>
<tr><td>start-tc-server</td><td>démarrer un serveur du cluster</td></tr>
<tr><td>stop-tc-server</td><td>arrêter un serveur du cluster</td></tr>
<tr><td>tc-stats</td><td>(Terracotta cluster statistics recorder) permet de configurer et de gérer l'enregistrement des statistiques dans le cluster</td></tr>
<tr><td>tim-get</td><td>gérer les modules installés de Terracotta (TIM)</td></tr>
<tr><td>version</td><td>afficher la version de Terracotta</td></tr>
</table>

47.3.4. Les limitations

Tous les éléments du cluster (clients et serveurs) doivent utiliser la même JVM (vendeur et version) et la même version de Terracotta.

Attention : toutes les JVM ne sont pas supportées. Vu le mode de fonctionnement impliquant des interactions de bas niveau avec la JVM, il n'est pas surprenant que Terracotta ne fonctionne qu'avec certaines implémentations de la JVM et versions de ces implémentations.

Terracotta peut gérer dans le cluster la plupart des objets Java mais il y a cependant des restrictions notamment des classes qui encapsulent une ressource externe (fichier, socket réseau, connexion vers des ressources, ...) ou spécifique à la JVM (Runtime, Thread, ...)

Un objet ne peut donc être géré par le cluster que s'il est portable. Toute tentative de faire gérer par le cluster un objet non portable lève une exception de type TCNonPortableObjectException.

47.4. Les cas d'utilisation

Terracotta propose du clustering d'objets Java au niveau de la JVM dans un but généraliste. Les cas d'utilisation sont donc nombreux parmi lesquels :

- partage de données entre les instances d'une même application
- avoir un singleton sur un cluster
- cache distribué de niveau 1 ou 2
- réplication de sessions http de conteneurs web de façon efficace
- partage de données entre plusieurs applications
- heap virtuel pour le partage de grande quantité de données avec lazy loading
- répartition de charge grâce au partage d'une queue des traitements à effectuer par les différentes JVM
- utilisation comme système de messaging, sans avoir à mettre en oeuvre une solution de type MOM, en permettant l'échange d'objets directement entre JVM
- partage de très grande quantité de données, supérieure à la capacité des clients : seules celles utilisées seront copiées localement
- ...

Certains de ces cas sont détaillés dans les sections suivantes.

47.4.1. La réplication de sessions HTTP

Chaque conteneur web propose sa propre implémentation concernant le stockage des sessions. Les données d'une session particulière sont retrouvées grâce à un identifiant unique nommé jsessionid.

La tendance est aux applications web stateless côté serveur mais cela implique que l'état soit stocké côté client (dans une application de type RIA) ou soit stocké dans la base de données mais ces deux solutions ne sont pas toujours possible à mettre en oeuvre. Si le contexte est stocké dans une session, ce qui est le cas dans beaucoup d'applications web, il est nécessaire de répliquer ces sessions dans les différents noeuds du cluster ne serait ce que pour garantir le fail-over.

Terracotta peut permettre d'être plus efficace dans les réplications des sessions d'un cluster de conteneurs web que l'utilisation des mécanismes proposés par ces derniers qui utilisent généralement la sérialisation. Le mode de fonctionnement de Terracotta peut aussi permettre l'utilisation d'une affinité de session : tant que le serveur répond, il n'y pas de réplication des données sur les autres noeuds du cluster. Dès que le serveur tombe, un autre noeud répond et va obtenir les données de la session du serveur Terracotta. Les échanges réseaux sont donc limités.

De plus les objets stockés dans la session n'ont pas l'obligation d'implémenter l'interface Serializable.

Fréquemment pour optimiser la réplication des sessions fournies par les conteneurs, les objets de la session sont répartis dans différents attributs. Le mécanisme optimisé de la réplication des objets proposés par Terracotta évite d'avoir à sérialiser le graphe d'objets.

Pour faciliter la configuration, Terracotta propose des configurations par défaut pour conteneurs (Tomcat, Jetty, JBoss, GlassFish, ...) qui vont permettre de gérés les objets de stockages de la session dans le cluster Terracotta.

La configuration est très simple puisqu'il suffit

1. d'utiliser le module adéquat pour le conteneur utilisé
2. de fournir le nom de la webapp dont les sessions doivent être gérées
3. de déclarer l'instrumentation des types d'objets qui seront mis dans la session

Exemple :

```
<tc:tc-config xmlns:tc="http://www.terracotta.org/config">

  ...

  <application>
    <dso>
      <instrumented-classes>
        <include>
          <class>com.jmdoudoux.test.terracotta.MaClasse</class>
        </include>
      </instrumented-classes>
      <web-applications>
        <web-application>MaWebApp</web-application>
      </web-applications>
    </dso>
  </application>
</tc:tc-config>
```

Pour limiter les échanges de données entre les noeuds du cluster, il peut être utile d'utiliser l'affinité de session au niveau du load balancer pour permettre que cela soit toujours le même noeud qui réponde à un même client dans des conditions d'utilisation normale.

47.4.2. Un cache distribué

La mise en cluster d'un graphe d'objets se prête particulièrement bien à une utilisation sous la forme d'un cache distribué. Ainsi les données du cache sont répliquées dans les différents noeuds du cluster avec un chargement unique dans un noeud et la réplication dans les autres évitant ainsi un chargement à chaque noeud.

Terracotta peut distribuer un cache dont il propose un TIM par exemple EhCache ou une solution maison plus simple utilisant par exemple une collection de type Map gérant les accès concurrents comme ConcurrentHashMap.

Si le cache est fréquemment mis à jour, il faut être vigilent sur le positionnement des verrous pour ne pas dégrader les performances d'accès au cache qui inhiberait l'intérêt de sa mise en oeuvre.

47.4.3. La répartition de charge de traitements

Ce cas d'utilisation met en oeuvre le motif de conception master-worker. Un unique master créé des tâches et les empiles dans une collection partagée généralement de type queue. Plusieurs workers se chargent de traiter une tâche qu'elle a dépilée.

Ceci permet de paralléliser les traitements de ces tâches dans les différentes JVM qui composent le cluster.

47.4.4. Le partage de données entre applications

Terracotta permet de partager des objets entre plusieurs instances d'une même application mais peut aussi permettre le partage d'objets entre différentes applications. Bien sûre, ces objets doivent avoir des caractéristiques communes notamment celle définies comme racine.

Ceci peut être très pratique pour par exemple partager des données entre une version standalone et une version web d'une application.

47.5. Quelques exemples de mise en oeuvre

Terracotta est fourni avec plusieurs applications d'exemples notamment une application de type chat, de partage de données, d'édition graphique, de type Queue, ...

Cette section va proposer quelques exemples simples de mise en ouvre de Terracotta. Dans un contexte concret, les principes utilisés sont les mêmes mais sont généralement plus complexes notamment en ce qui concerne la définition du contenu du fichier de configuration.

47.5.1. Un exemple simple avec une application standalone

L'application utilisée dans cet exemple incrémente simplement un compteur static.

Exemple :

```
package com.jmdoudoux.test.terracotta;

public class MainTest {

  private static int compteur;

  public static void main(String[] args)
  {
      compteur++;
      System.out.println("Compteur = " + compteur);
  }

}
```

Remarque : cet exemple est basique car il ne gère pas les accès concurrents à la variable compteur. Il ne peut donc fonctionner correctement que si une seule instance de l'application est en cours d'exécution.

Chaque exécution de cette application affiche toujours la valeur 1 pour le compteur puisque sa valeur est initialisée à chaque lancement d'une JVM.

Résultat :

```
C:\eclipse34\workspace\TestTerracotta\bin>java -cp . com.jmdoudoux.test.terracot
ta.MainTest
Compteur = 1

C:\eclipse34\workspace\TestTerracotta\bin>java -cp . com.jmdoudoux.test.terracot
ta.MainTest
Compteur = 1
```

La mise en cluster de cette application va permettre de maintenir la valeur du compteur dans le cluster et permettre ainsi son incrémentation à chaque exécution.

Le fichier de configuration de Terracotta contient uniquement la définition d'une racine qui correspond au compteur.

Exemple :

```
<tc:tc-config xmlns:tc="http://www.terracotta.org/config">
  <application>
    <dso>
      <roots>
        <root>
          <field-name>com.jmdoudoux.test.terracotta.MainTest.compteur</field-name>
        </root>
```

```
      </roots>
    </dso>
  </application>
</tc:tc-config>
```

Il faut lancer le serveur Terracotta dans une boite de commande dédiée.

Résultat :

```
C:\>start-tc-server.bat
2010-04-25 21:29:58,555 INFO - Terracotta 3.2.0, as of 20100107-130122 (Revision
 14244 by cruise@su10mo5 from 3.2)
2010-04-25 21:30:00,040 INFO - Configuration loaded from the Java resource at '/
com/tc/config/schema/setup/default-config.xml', relative to class com.tc.config.
schema.setup.StandardXMLFileConfigurationCreator.
2010-04-25 21:30:01,009 INFO - Log file: 'C:\Documents and Settings\jmd\terracot
ta\server-logs\terracotta-server.log'.
2010-04-25 21:30:04,383 INFO - Available Max Runtime Memory: 504MB
2010-04-25 21:30:08,305 INFO - JMX Server started. Available at URL[service:jmx:
jmxmp://0.0.0.0:9520]
2010-04-25 21:30:10,399 INFO - Terracotta Server instance has started up as ACTI
VE node on 0.0.0.0:9510 successfully, and is now ready for work.
```

Il faut ensuite exécuter l'application dans une JVM configurée pour être utilisable dans le cluster. Pour une application standalone, Terracotta propose le script dso-java qui lance une JVM qui est configurée pour devenir un client du cluster.

Par défaut, l'outil dso-java utilise le fichier tc-config.xml présent dans le répertoire courant. Il est possible de préciser un autre fichier en utilisant l'option -Dtc.config

Résultat :

```
C:\eclipse34\workspace\TestTerracotta\bin>dso-java -cp . com.jmdoudoux.test.terr
acotta.MainTest
 Starting Terracotta client...
2010-04-25 21:42:00,615 INFO - Terracotta 3.2.0, as of 20100107-130122 (Revision
 14244 by cruise@su10mo5 from 3.2)
2010-04-25 21:42:01,891 INFO - Configuration loaded from the file at 'C:\eclipse
34\workspace\TestTerracotta\bin\tc-config.xml'.
2010-04-25 21:42:02,343 INFO - Log file: 'C:\eclipse34\workspace\TestTerracotta\
bin\logs-172.16.0.50\terracotta-client.log'.
2010-04-25 21:42:13,204 INFO - Connection successfully established to server at
127.0.0.1:9510
Compteur = 1

C:\eclipse34\workspace\TestTerracotta\bin>dso-java -cp . com.jmdoudoux.test.terr
acotta.MainTest
 Starting Terracotta client...
2010-04-25 21:42:18,167 INFO - Terracotta 3.2.0, as of 20100107-130122 (Revision
 14244 by cruise@su10mo5 from 3.2)
2010-04-25 21:42:18,728 INFO - Configuration loaded from the file at 'C:\eclipse
34\workspace\TestTerracotta\bin\tc-config.xml'.
2010-04-25 21:42:18,852 INFO - Log file: 'C:\eclipse34\workspace\TestTerracotta\
bin\logs-172.16.0.50\terracotta-client.log'.
2010-04-25 21:42:20,906 INFO - Connection successfully established to server at
127.0.0.1:9510
Compteur = 2

C:\eclipse34\workspace\TestTerracotta\bin>dir
 Volume in drive C has no label.
 Volume Serial Number is 1B46-3C32

 Directory of C:\eclipse34\workspace\TestTerracotta\bin

25/04/2010  21:42    <DIR>          .
25/04/2010  21:42    <DIR>          ..
25/04/2010  21:22    <DIR>          com
25/04/2010  21:42    <DIR>          logs-127.0.0.1
25/04/2010  21:23               283 tc-config.xml
```

```
C:\eclipse34\workspace\TestTerracotta\bin>
```

L'état de la variable gérée par le cluster est automatiquement fourni au client une fois créée, ce qui permet une incrémentation sans réinitialisation par les clients.

47.5.2. Un second exemple avec une application standalone

Ce second exemple va utiliser une application qui stocke sa date/heure d'exécution dans une collection, attend 10 secondes et affiche le contenu de la collection.

Exemple :

```
package com.jmdoudoux.test.terracotta;

import java.util.*;

public class MainTest1 {

  private List<String> donnees = new ArrayList<String>();

  public void traiter() {
    synchronized (donnees) {
      donnees.add("Traitement du " + new Date());
      try {
        Thread.sleep(10000);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
      for (String donnee : donnees) {
        System.out.println(donnee);
      }
    }
  }

  public static void main(String[] args) {
    System.out.println("Demarrage de l'application");
    new MainTest1().traiter();
    System.out.println("Arret de l'application");
  }

}
```

A chaque exécution de cette application, la collection ne contient que l'occurrence du traitement courant puisque la collection est recréée dans chaque JVM.

Il est possible avec Terracotta de maintenir l'état de la collection sur le serveur Terracotta et ainsi de partager l'objet entre différentes instances de JVM qui exécutent l'application en concomitance ou non.

Ce qui est intéressant avec Terracotta c'est que le code de l'application n'est pas modifié : aucune API de Terracotta n'est utilisée. Seul l'environnement d'exécution est différent.

Il faut tout d'abord créer un fichier de configuration pour Terracotta nommé tc-config.xml

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<tc:tc-config xmlns:tc="http://www.terracotta.org/config">
  <application>
    <dso>
      <roots>
        <root>
          <field-name>com.jmdoudoux.test.terracotta.MainTest1.donnees
```

```
          </field-name>
        </root>
      </roots>
      <locks>
        <autolock>
          <method-expression>
            * com.jmdoudoux.test.terracotta.MainTest1*.*(..)
          </method-expression>
            <lock-level>write</lock-level>
        </autolock>
      </locks>
      <instrumented-classes>
        <include>
          <class-expression>
            com.jmdoudoux.test.terracotta.MainTest1
          </class-expression>
        </include>
      </instrumented-classes>
    </dso>
  </application>
</tc:tc-config>
```

Ce fichier permet de préciser quelles sont les classes qui sont prises en charge par Terracotta.

Il faut lancer le serveur Terracotta en exécutant la commande start-tc-server

Résultat :

```
C:\Documents and Settings\jmd>start-tc-server.bat
2010-02-12 21:00:08,599 INFO - Terracotta 3.2.0, as of 20100107-130122 (Revision
 14244 by cruise@su10mo5 from 3.2)
2010-02-12 21:00:09,193 INFO - Configuration loaded from the Java resource at '/
com/tc/config/schema/setup/default-config.xml', relative to class com.tc.config.
schema.setup.StandardXMLFileConfigurationCreator.
2010-02-12 21:00:09,490 INFO - Log file: 'C:\Documents and Settings\jmd\terracot
ta\server-logs\terracotta-server.log'.
2010-02-12 21:00:10,068 INFO - Available Max Runtime Memory: 504MB
2010-02-12 21:00:11,130 INFO - JMX Server started. Available at URL[service:jmx:
jmxmp://0.0.0.0:9520]
2010-02-12 21:00:12,287 INFO - Terracotta Server instance has started up as ACTI
VE node on 0.0.0.0:9510 successfully, and is now ready for work.
```

Il faut exécuter l'application en utilisant la commande dso-java

Résultat :

```
C:\eclipse34\workspace\TestTerracotta\bin>dso-java com.jmdoudoux.test.terracotta.MainTest1
 Starting Terracotta client...
2010-02-12 21:04:20,410 INFO - Terracotta 3.2.0, as of 20100107-130122 (Revision
 14244 by cruise@su10mo5 from 3.2)
2010-02-12 21:04:21,019 INFO - Configuration loaded from the file at 'C:\eclipse
34\workspace\TestTerracotta\bin\tc-config.xml'.
2010-02-12 21:04:21,144 INFO - Log file: 'C:\eclipse34\workspace\TestTerracotta\
bin\logs-127.0.0.1\terracotta-client.log'.
2010-02-12 21:04:24,097 INFO - Connection successfully established to server at 127.0.0.1:9510
Demarrage de l'application
Hello, World Fri Feb 12 21:04:24 CET 2010
Arret de l'application
```

Pour tirer avantages de Terracotta, il faut ouvrir deux boites de commandes et exécuter deux fois l'application en simultanée.

Si l'application est exécutée une quatrième fois, les données des trois premières exécutions sont toujours présentes tant que le serveur Terracotta n'est pas arrêté.

47.5.3. Un exemple avec une application web

L'exemple de cette section va exécuter une application sur deux instances d'un serveur Tomcat version 6.0 mis en cluster.

Cette webapp contient une unique servlet qui stocke sa date/heure d'invocation dans un singleton qui encapsule une collection et affiche le contenu de la collection.

Le code du singleton est plutôt basic.

Exemple :

```
package com.jmdoudoux.test.terracotta.web;

import java.util.*;

public class MonCache {
  private static MonCache instance = new MonCache();
  private List<String> maListe;

  private MonCache() {
    maListe = new ArrayList<String>();
  }

  public static MonCache getInstance() {
    return instance;
  }

  public List<String> getDonnees() {
    return Collections.unmodifiableList(maListe);
  }

  public synchronized void ajouter(String occurence) {
    maListe.add(occurence);
  }

  public synchronized void effacer() {
    maListe.clear();
  }
}
```

Cette classe gère les accès concurrents notamment au niveau des méthodes qui opèrent des mises à jour dans la collection et elle renvoie une version immuable de la collection. Ces éléments sont importants pour la bonne mise en oeuvre de Terracotta.

Il faut obtenir et installer le TIM dédié à Tomcat 6.0 en utilisant la commande tim-get avec l'option install suivi du module à installer et de se version.

Résultat :

```
C:\Users\Jean Michel>tim-get.bat list
Terracotta 3.2.0, as of 20100107-130122 (Revision 14244 by cruise@su10mo5 from 3
.2)

*** Terracotta Integration Modules for TC 3.2.0 ***

(+) ehcache-terracotta 1.8.0 [net.sf.ehcache]
(+) terracotta-hibernate-cache 1.1.0 [org.terracotta.hibernate]
(-) pojoizer 1.0.4
(-) tim-annotations 1.5.0
(-) tim-apache-collections-3.1 1.2.0
...
(-) tim-tomcat-5.0 2.1.0
(-) tim-tomcat-5.5 2.1.0
(-) tim-tomcat-6.0 2.1.0
(-) tim-vector 2.6.1
(-) tim-wan-collections 1.1.0
(-) tim-weblogic-10 2.1.0
(-) tim-weblogic-9 2.1.0
```

```
(-) tim-wicket-1.3 1.4.0
(+) quartz-terracotta 1.0.0 [org.terracotta.quartz]
(-) simulated-api 1.2.0 [org.terracotta]

(+) Installed  (-) Not installed  (!) Installed but newer version exists

C:\Users\Jean Michel>tim-get install tim-tomcat-6.0 2.1.0
Terracotta 3.2.0, as of 20100107-130122 (Revision 14244 by cruise@su10mo5 from 3
.2)

Installing tim-tomcat-6.0 2.1.0 and dependencies...
   INSTALLED: tim-tomcat-6.0 2.1.0 - Ok
   INSTALLED: tim-tomcat-5.5 2.1.0 - Ok
   INSTALLED: tim-tomcat-common 2.1.0 - Ok
   SKIPPED: tim-session-common 2.1.0 - Already installed

Done. (Make sure to update your tc-config.xml with the new/updated version if ne
cessary)

C:\Users\Jean Michel>
```

Il faut fournir en paramètre de la JVM les informations utiles à l'agent de Terracotta. Le plus simple est d'utiliser le script dso-env fourni par Terracotta :

- Définir la variable d'environnement TC_INSTALL_DIR
- Définir la variable d'environnement TC_CONFIG_PATH
- Invoquer le script dso-env contenu dans le répertoire bin de Terracotta
- Ajouter la variable TC_JAVA_OPTS à la variable JAVA_OPTS dans le script catalina de Tomcat lorsque celui-ci est utilisé pour démarrer Tomcat.

Exemple sous Linux/Unix :

Résultat :

```
...
export TC_INSTALL_DIR=<chemin_rep_Terracotta>
export TC_CONFIG_PATH=<chemin_fichier_tc-config.xml>
. $TC_INSTALL_DIR/bin/dso-env.sh -q
export JAVA_OPTS="$TC_JAVA_OPTS $JAVA_OPTS"
...
```

Exemple sous Windows :

Résultat :

```
...
set TC_INSTALL_DIR=< chemin_rep_Terracotta >
set TC_CONFIG_PATH=<path_to_local_tc-config.xml>
%TC_INSTALL_DIR%\bin\dso-env.bat -q
set JAVA_OPTS=%TC_JAVA_OPTS%;%JAVA_OPTS%
...
```

Il faut lancer le serveur Terracotta en lançant le script start-tc-server puis les serveurs du cluster.

La suite de cette section sera développée dans une version future de ce document

47.5.4. La réplication de sessions sous Tomcat

Cet exemple va utiliser Terracotta pour assurer la réplication de sessions dans un cluster de serveurs Tomcat

La suite de cette section sera développée dans une version future de ce document

47.5.5. Le partage de données entre deux applications

Cet exemple va permettre à deux applications d'incrémenter une variable commune partagée par Terracotta. Cette variable est encapsulée dans un objet propre à chacune des applications.

Exemple :

```
package com.jmdoudoux.test.terracotta.appli_a;

public class MonObjetA {

  private static int compteur;

  public static synchronized int incrementer() {
    compteur++;
    return compteur;
  }
}
```

La première application incrémente et affiche la variable deux fois.

Exemple :

```
package com.jmdoudoux.test.terracotta.appli_a;

public class AppliA {

  public static void main(String[] args) {

    System.out.println("Appli_A compteur="+MonObjetA.incrementer());

    System.out.println("Appli_A compteur="+MonObjetA.incrementer());
  }

}
```

Le fichier de configuration de Terracotta pour l'application ne dispose que d'une seule particularité : la racine du compteur partagé est défini avec un nom qui permettra d'y faire référence dans le fichier de configuration de l'autre application. Ce nom permettra à Terracotta d'identifier les deux objets comme étant le même : toutes les modifications dans l'un seront reportées dans l'autre et vice et versa.

Exemple :

```
<tc:tc-config xmlns:tc="http://www.terracotta.org/config">
  <application>
    <dso>
      <roots>
```

```
        <root>
          <field-name>com.jmdoudoux.test.terracotta.appli_a.MonObjetA.compteur</field-name>
          <root-name>MonCompteur</root-name>
        </root>
      </roots>
      <locks>
        <autolock>
          <method-expression>
            * com.jmdoudoux.test.terracotta.appli_a.MonObjetA.*()
          </method-expression>
            <lock-level>write</lock-level>
        </autolock>
      </locks>
      <instrumented-classes>
        <include>
          <class-expression>
            com.jmdoudoux.test.terracotta.appli_a.*
          </class-expression>
        </include>
      </instrumented-classes>
    </dso>
  </application>
</tc:tc-config>
```

La seconde application encapsule aussi une variable statique de type int dans un objet dédié.

Exemple :

```
package com.jmdoudoux.test.terracotta.appli_b;

public class AppliB {
  public static void main(String[] args) {

    System.out.println("Appli_B compteur="+MonObjetB.incrementer());
  }
}
```

La seconde application va incrémenter le compteur et afficher la valeur.

Exemple :

```
package com.jmdoudoux.test.terracotta.appli_b;

public class MonObjetB {

  private static int compteur;

  public static synchronized int incrementer() {
    compteur++;
    return compteur;
  }
}
```

Le fichier de configuration est similaire à celui de la première application en utilisant ses propres objets.

Exemple :

```
<tc:tc-config xmlns:tc="http://www.terracotta.org/config">
  <application>
    <dso>
      <roots>
        <root>
          <field-name>com.jmdoudoux.test.terracotta.appli_b.MonObjetB.compteur</field-name>
          <root-name>MonCompteur</root-name>
        </root>
      </roots>
```

```
      <locks>
        <autolock>
          <method-expression>
            * com.jmdoudoux.test.terracotta.appli_b.MonObjetB.*()
          </method-expression>
            <lock-level>write</lock-level>
        </autolock>
      </locks>
      <instrumented-classes>
        <include>
          <class-expression>
            com.jmdoudoux.test.terracotta.appli_b.*
          </class-expression>
        </include>
      </instrumented-classes>
    </dso>
  </application>
</tc:tc-config>
```

Avant de lancer les applications, il faut lancer le serveur Terracotta.

Exemple :

```
C:\>start-tc-server.bat
2010-04-25 14:38:03,139 INFO - Terracotta 3.2.0, as of 20100107-130122 (Revision
 14244 by cruise@su10mo5 from 3.2)
2010-04-25 14:38:03,733 INFO - Configuration loaded from the Java resource at '/
com/tc/config/schema/setup/default-config.xml', relative to class com.tc.config.
schema.setup.StandardXMLFileConfigurationCreator.
2010-04-25 14:38:04,154 INFO - Log file: 'C:\Documents and Settings\jmd\terracot
ta\server-logs\terracotta-server.log'.
2010-04-25 14:38:06,701 INFO - Available Max Runtime Memory: 504MB
2010-04-25 14:38:09,279 INFO - JMX Server started. Available at URL[service:jmx:
jmxmp://0.0.0.0:9520]
2010-04-25 14:38:10,264 INFO - Terracotta Server instance has started up as ACTI
VE node on 0.0.0.0:9510 successfully, and is now ready for work.
```

Pour vérifier le partage de la donnée entre les deux applications, le test suivant est exécuté :

- Lancer l'application A
- Lancer l'application B
- Lancer l'application B
- Lancer l'application A

Résultat :

```
C:\eclipse34\workspace\TestTerracottaA\bin>dso-java -cp . com.jmdoudoux.test.ter
racotta.appli_a.AppliA
Starting Terracotta client...
2010-04-25 14:40:12,821 INFO - Terracotta 3.2.0, as of 20100107-130122 (Revision
14244 by cruise@su10mo5 from 3.2)
2010-04-25 14:40:13,414 INFO - Configuration loaded from the file at 'C:\eclipse
34\workspace\TestTerracottaA\bin\tc-config.xml'.
2010-04-25 14:40:13,586 INFO - Log file: 'C:\eclipse34\workspace\TestTerracottaA
\bin\logs-172.16.0.50\terracotta-client.log'.
2010-04-25 14:40:15,805 INFO - Connection successfully established to server at
172.16.0.50:9510
Appli_A compteur=1
Appli_A compteur=2
C:\eclipse34\workspace\TestTerracottaB\bin>dso-java -cp . com.jmdoudoux.test.ter
racotta.appli_b.AppliB
Starting Terracotta client...
2010-04-25 14:40:28,226 INFO - Terracotta 3.2.0, as of 20100107-130122 (Revision
14244 by cruise@su10mo5 from 3.2)
2010-04-25 14:40:28,820 INFO - Configuration loaded from the file at 'C:\eclipse
34\workspace\TestTerracottaB\bin\tc-config.xml'.
2010-04-25 14:40:28,960 INFO - Log file: 'C:\eclipse34\workspace\TestTerracottaB
\bin\logs-172.16.0.50\terracotta-client.log'.
```

```
2010-04-25 14:40:30,976 INFO - Connection successfully established to server at
172.16.0.50:9510
Appli_B compteur=3

C:\eclipse34\workspace\TestTerracottaB\bin>dso-java -cp . com.jmdoudoux.test.ter
racotta.appli_b.AppliB
Starting Terracotta client...
2010-04-25 14:40:35,757 INFO - Terracotta 3.2.0, as of 20100107-130122 (Revision
14244 by cruise@su10mo5 from 3.2)
2010-04-25 14:40:36,351 INFO - Configuration loaded from the file at 'C:\eclipse
34\workspace\TestTerracottaB\bin\tc-config.xml'.
2010-04-25 14:40:36,491 INFO - Log file: 'C:\eclipse34\workspace\TestTerracottaB
\bin\logs-172.16.0.50\terracotta-client.log'.
2010-04-25 14:40:38,476 INFO - Connection successfully established to server at
172.16.0.50:9510
Appli_B compteur=4
C:\eclipse34\workspace\TestTerracottaA\bin>dso-java -cp . com.jmdoudoux.test.ter
racotta.appli_a.AppliA
Starting Terracotta client...
2010-04-25 14:42:27,986 INFO - Terracotta 3.2.0, as of 20100107-130122 (Revision
14244 by cruise@su10mo5 from 3.2)
2010-04-25 14:42:28,580 INFO - Configuration loaded from the file at 'C:\eclipse
34\workspace\TestTerracottaA\bin\tc-config.xml'.
2010-04-25 14:42:28,752 INFO - Log file: 'C:\eclipse34\workspace\TestTerracottaA
\bin\logs-172.16.0.50\terracotta-client.log'.
2010-04-25 14:42:30,768 INFO - Connection successfully established to server at
172.16.0.50:9510
Appli_A compteur=5
Appli_A compteur=6
```

Le compteur est incrémenté correctement selon les invocations des applications.

47.6. La console développeur

Terracotta propose un outil particulièrement utile pour obtenir des informations sur le cluster et le monitorer : la console développeur (Developer Console).

Cet outil permet de voir de nombreuses informations sur l'état d'un cluster, notamment :

- Consulter l'état des objets gérés par le cluster
- Obtenir des métriques sur les activités du cluster (nombre d'objets instanciés, nombre de transactions réalisées, ...)

A partir de ces informations, il est possible de déterminer des actions pour améliorer les performances notamment sur les verrous.

47.7. Le fichier de configuration

Comme Terracotta ne propose aucune API, la configuration se fait dans un fichier de configuration qui est utilisé par le serveur et les noeuds du cluster.

Le fichier de configuration de Terracotta est un fichier XML qui permet de décrire les caractéristiques et le comportement du ou serveurs Terracotta et des clients.

Le fichier de configuration est un document XML qui décrit :

- la liste et les paramètres des serveurs
- la configuration des clients
- la liste des classes à instrumenter
- les racines des graphes d'objets gérées par le cluster

* les verrous

Au lancement d'un serveur, celui ci recherche le fichier de configuration qui peut être

* le fichier de configuration par défaut fournit avec Terracotta : celui-ci est utilisé si aucun fichier n'est précisé et si aucun fichier tc-config.xml n'est présent dans le répertoire où le serveur est lancé
* un fichier local : soit précisé explicitement avec l'option -f de la commande start-tc-server soit le fichier tc-config.xml dans le répertoire courant où le serveur est lancé
* un fichier distant : l'url doit être précisée explicitement avec l'option -f de la commande start-tc-server

Au lancement d'un client, celui ci recherche le fichier de configuration qui peut être :

* un fichier local : soit précisé explicitement avec l'option -f de la commande start-tc-server soit le fichier tc-config.xml dans le répertoire courant où le serveur est lancé
* un fichier distant : l'url doit être précisée explicitement avec l'option -f de la commande start-tc-server
* obtenu à partir d'un serveur : cette solution permet de garantir la synchronisation du fichier de configuration entre le serveur de client puisque le fichier n'est déployé que sur le serveur

Le contenu du fichier de configuration chargé par un serveur ou un client est inséré dans le fichier de log. Il est aussi consultable grâce à la console du développeur.

Le fichier XML de configuration est composé de plusieurs éléments principaux facultatifs :

* system : permet de configurer des éléments de tous les composants du cluster
* servers : permet de configurer le ou les serveurs du cluster
* clients : permet de configurer le ou les noeuds du cluster
* application : permet de configurer les éléments qui seront gérés par le cluster

47.7.1. La configuration de la partie system

Cette partie est définie dans le tag <system>.

Il possède un tag fils <configuration-model> qui permet de préciser un modèle de configuration : les valeurs possibles sont development ou production.

Par défaut, c'est le modèle development qui est utilisé. Il permet à chaque client d'avoir son propre fichier de configuration.

Avec la modèle production, les clients obtiennent le fichier de configuration d'un serveur, ce qui assure une cohérence entre la configuration des éléments du cluster (serveurs et clients). La JVM d'un noeud client doit avoir la propriété tc.config valorisée avec le nom du serveur et son port : -Dtc.config=host:port

47.7.2. La configuration de la partie serveur

Cette partie permet de configurer le ou les serveurs du cluster. Elle est définie dans le tag <servers>.

Si cette section est omise, alors le cluster contiendra un seul noeud avec les valeurs par défaut.

Chaque serveur du cluster est défini dans un tag fils <server>. Ce tag possède plusieurs attributs :

Attribut	Rôle
host	host du serveur Valeur par défaut : l'adresse IP de la machine locale
name	nom servant d'identifiant du serveur
bind	Valeur par défaut : 0.0.0.0

Si plusieurs serveurs sont définis dans le fichier de configuration, chaque serveur doit être démarré en précisant le nom du serveur.

Le tag fils <data> permet de préciser le répertoire qui va contenir les données du serveur stockées sur le système de fichier

Le tag fils <logs> permet de préciser le répertoire qui va contenir les logs du serveur.

Chaque serveur doit avoir un répertoire de logs distinct qui par défaut est le sous répertoire terracotta/server-logs du répertoire de l'utilisateur.

Le tag fils <statistics> permet de préciser le répertoire qui va contenir les données statistiques du serveur.

Le tag fils <dso-port> permet de préciser le port d'écoute du serveur qui par défaut est le port 9510.

Le tag fils <jmx-port> permet de préciser le port JMX du serveur qui par défaut est le port 9520.

Le tag fils <l2-group-port> permet de préciser le port utilisé pour la communication entre les serveurs en mode actif-passif qui par défaut est le port 9530.

Le tag <dso> permet de configurer le service DSO du serveur.

Le tag fils <client-reconnect-window> permet de définir un temps de reconnexion pour un client exprimé en secondes. La valeur par défaut est 120.

Le tag fils <persistence> permet de préciser si les données gérés par le cluster sont persistantes ou non. Les valeurs possibles sont :

- temporary-swap-only : les données peuvent être temporairement écrites sur le système de fichiers mais elles ne survivent pas à un arrêt du cluster
- permanent-store : toutes les données gérées par le cluster sont stockées sur le système de fichiers ce qui permet de mettre en oeuvre un failover sur le cluster

Le tag fils <garbage-collection> permet de configurer le ramasse miette distribué. Il possède plusieurs tags fils :

Tag	Rôle
enabled	Activer ou non le DGC. La désactivation n'est utile que si aucune des instances gérées par le cluster n'est supprimée Par défaut : true
verbose	Activer ou non l'inclusion des informations relatives aux activités du DGC dans la log Par défaut : false
interval	Préciser le temps d'attente entre deux exécutions du DGC en secondes Par défaut : 3600

Le tag <ha> permet de configurer le mode fonctionnement des serveurs du cluster : il faut donc qu'il y ait au moins 2 serveurs de configurés.

Le tag fils <mode> peut prendre deux valeurs :

- networked-active-passive :
- disk-based-active-passive :

Le tag fils <mirror-groups> permet de définir des groupes de serveurs.

47.7.3. La configuration de la partie cliente

Cette partie permet de configurer le ou les clients du cluster. Elle est définie dans le tag <clients>.

Exemple :

```
<clients>
    <logs>/home/logs/terracotta/client-logs
    </logs>
    <statistics>/home/logs/terracotta/client-stats
    </statistics>
    <modules>
        <module name="tim-tomcat-6.0" version="2.1.0" />
        <module name="tim-vector" version="2.6.0" />
        <module name="tim-hashtable" version="2.6.0" />
    </modules>
</clients>
```

Le tag fils <logs> permet de préciser où seront stockés les fichiers de logs.

Chaque client doit avoir un répertoire de logs distinct qui par défaut est le sous répertoire terracotta/client-logs du répertoire de l'utilisateur.

Le tag fils <statistics> permet de préciser le répertoire qui va contenir les données statistiques du client.

Le tag fils <modules> permet de définir les TIM qui seront utilisés par les clients. Le tag <modules> contient un tag fils <module> pour chaque TIM utilisé.

Le tag <module> possède plusieurs attributs :

- name : contient le nom du module
- version : contient le numéro de version du module
- group-id : contient le nom du package du module (requis si le module n'est pas dans le répertoire par défaut des modules)

Terracotta recherche un jar correspondant au module à partir des informations fournies par défaut dans le sous répertoire modules du répertoire d'installation de Terracotta. Il est possible de préciser des répertoires supplémentaires en utilisant le tag <repository> fils du tag <modules>. Si un module est stocké dans un de ces répertoires, l'attribut group-id du module doit être précisé.

47.7.4. La configuration de la partie applicative

Cette partie permet de configurer le ou les éléments à gérer par le cluster. Elle est définie dans le tag <dso> du tag fils <application>.

La configuration de ces éléments se fait au travers de trois entités principales :

- les racines (roots)
- les verrous (locks)
- les classes à instrumenter (instrumented classes)

47.7.4.1. La définition des racines

Les racines permettent de définir l'objet qui sera géré par le cluster en tant qu'objet père d'une hiérarchie d'objets composés de ces dépendances.

Les racines sont précisées dans un tag <roots> fils du tag application/dso.

Chaque objet racine est défini dans un tag <root>.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<tc:tc-config xmlns:tc="http://www.terracotta.org/config">
...
  <application>
    <dso>
      <roots>
        <root>
          <field-name>com.jmdoudoux.test.terracotta.MaClasse.monChamp
          </field-name>
        </root>
      </roots>
...
    </dso>
  </application>
</tc:tc-config>
```

Le tag fils <field-name> permet de préciser le champ d'une classe qui sera la racine du graphe d'objets gérés par le cluster. Ce graphe contient les dépendances de l'objet racine.

Le tag fils <root-name> permet de fournir un identifiant à la racine permettant d'y faire référence.

47.7.4.2. La définition des verrous

Les verrous sont précisés dans un tag <autolock> fils du tag application/dso/locks.

Il existe deux types de verrous :

- Autolock : permet de propager les instructions de synchronisation (méthodes ou blocs synchronized, wait(), notify(), ...) au travers du cluster
- Named lock : permet de définir un verrou nominatif

Ces verrous définissent les méthodes qui posent des verrous afin de permettre à Terracotta de les propager à tous les noeuds du cluster, assurant ainsi une gestion sécurisée et distribuée des accès concurrents.

Cette propagation est assurée par des traitements ajoutés lors de l'instrumentation : les verrous indiquent à Terracotta qu'il faut instrumenter le code de la méthode pour prendre en charge des verrous distribués permettant ainsi de gérer les accès concurrents sur les objets.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<tc:tc-config xmlns:tc="http://www.terracotta.org/config">
...
  <application>
...
    <dso>
      <locks>
        <autolock>
          <method-expression>* com.jmdoudoux.test.terracotta.MaClasse.get*(..)
          </method-expression>
            <lock-level>read</lock-level>
        </autolock>
        <autolock>
          <method-expression>* com.jmdoudoux.test.terracotta.MaClasse.set*(..)
          </method-expression>
            <lock-level>write</lock-level>
        </autolock>
      </locks>
...
    </dso>
  </application>
```

```
</tc:tc-config>
```

Le tag fils <autolock> permet de définir un autolock.

L'attribut auto-synchronized permet de rajouter dynamiquement au chargement de la classe le modificateur synchronized sur la méthode. Ceci est pratique pour éviter de modifier le code source ou si le code source n'est pas modifiable. L'utilisation de cette option est cependant à utiliser avec parcimonie et pertinence.

Un autolock sur une méthode ou un ensemble de méthodes est défini grâce au tag fils <method-expression> en utilisant une syntaxe particulière empruntée à AspectWerkz.

Résultat :

```
* com.jmdoudoux.test.terracotta.MaClasse.get*(..)
* com.jmdoudoux.test.terracotta.MaClasse.set*(..)
void com.jmdoudoux.test.terracotta.MaClasse.setChamp(java.lang.String)
void *..*(..)
```

Important : il faut limiter le nombre de méthodes à instrumenter pour ne pas dégrader les performances au chargement des classes et au runtime.

Le tag fils <lock-level> permet de définir le niveau du verrou. Quatre niveaux sont définis : write, read, concurrent et synchronous-write. Le niveau par défaut est write.

47.7.4.3. La définition des classes à instrumenter

Les classes à instrumenter sont celles :

- Dont les instances sont gérées par le cluster (objets racines ou dépendances)
- Qui utilisent des instances gérées par le cluster
- Qui posent des verrous distribués

L'instrumentation des classes est définie dans le tag fils <instrumented-classes>.

Le tag fils <include> permet de définir les classes à instrumenter et le tag fils <exclude> permet de définir les classes à ne pas instrumenter.

Le tag <class-expression> fils du tag <include> permet de définir un ensemble de classes en utilisant une syntaxe particulière empruntée à AspectWerkz.

Résultat :

```
com.jmdoudoux.test.terracotta.MaClasse
com.jmdoudoux.test.terracotta.Ma*
com.jmdoudoux.test.terracotta.*
com.jmdoudoux.*.terracotta.*
com.jmdoudoux.test..
```

Le tag <exclude> utilise la même syntaxe.

47.7.4.4. La définition des champs qui ne doivent pas être gérés

Le tag fils <transient-fields> permet de définir des champs d'objets gérés par le cluster qui ne doivent pas gérés par le cluster.

C'est notamment pratique pour des champs qui ne sont pas marqués avec le modificateur transient dans le code source.

Chaque champ est défini grâce à un tag <field-name>.

47.7.4.5. La définition des méthodes dont l'invocation doit être distribuée

Le tag fils <distributed-methods> permet de définir des méthodes d'objets gérés par le cluster qui doivent être invoquées dans tous les noeuds du cluster dès qu'elles sont invoquées dans un des noeuds.

Chaque méthode est définie grâce à un tag <method-expression>.

L'attribut run-on-all-nodes permet de préciser dans quel noeud l'invocation sera faite : true (valeur par défaut) demande l'invocation dans tous les noeuds, false demande l'invocation dans tous les noeuds qui possèdent déjà une référence sur l'objet.

47.7.4.6. La définition des webapps dont la session doit être gérée

Le tag fils <web-applications> permet de définir une ou plusieurs applications de type web dont le contenu de la session sera géré par le cluster

Chaque application web doit être ajoutée avec un tag <web-application> qui contient son nom.

47.8. La fiabilisation du cluster

Terracotta propose plusieurs fonctionnalités qui peuvent être combinées selon les besoins pour assurer la fiabilité, la disponibilité et la montée en charge du cluster :

- La fiabilité grâce à la persistance des données
- La haute disponibilité avec la redondance des serveurs
- La montée en charge avec la définition de groupes de serveurs

A son lancement, une JVM client se connecte au serveur Terracotta actif pour interagir avec lui. Dans un cluster avec un serveur actif, il est possible de définir un ou plusieurs serveurs passifs. Si le serveur actif est arrêté, un des serveurs passif est promu actif. Les clients du cluster tentent alors de se connecter au serveur actif parmi les serveurs configurés.

Si un serveur passif est configuré, l'arrêt du serveur actif est transparent pour les clients puisque ce serveur prend le relai en tant que serveur actif : les clients tentent de se connecter à ce serveur une fois qu'il est devenu actif. De nouveaux clients peuvent toujours rejoindre le cluster.

Si aucun serveur passif n'est configuré, le cluster est inutilisable tant que le serveur n'est pas redémarré. Les clients connectés avant l'arrêt attendent de pouvoir se reconnecter. Si le serveur était en mode persistant, les données sont restaurées et les clients peuvent se reconnecter. Si le serveur était en mode non persistant, les données sont perdues et les clients ne peuvent pas se reconnecter, ils doivent être arrêtés et démarrés pour se connecter au cluster.

Si un client ne peut se connecter à aucun serveur, il reste en attente tant qu'il n'arrive pas à se reconnecter à un serveur du cluster durant un temp configurable.

Un serveur peut être configuré pour être :

- Non persistant : les données des objets gérés par le cluster peuvent être écrites sur disque en cas de manque de mémoire sur le serveur. Les données sont perdues en cas d'arrêt et de redémarrage du cluster.
- Persistant : toutes les modifications sont écrites sur disque. Les données sont restaurées en cas d'arrêt et de redémarrage du cluster. Ce mode garantit que le redémarrage du cluster se fasse sans perte de données.

Si plusieurs serveurs sont utilisés en mode persistant, ils doivent être configurés pour écrire leurs données dans un système de fichiers partagés qui soit capable de gérer les accès concurrents en posant des verrous.

Il est possible d'utiliser et de configurer plusieurs serveurs Terracotta afin d'assurer une haute disponibilité du cluster Terracotta. Celle ci repose sur plusieurs fonctionnalités selon le niveau de sûreté souhaité :

- Fail over avec un serveur actif et un ou plusieurs serveurs passifs
- Persistance des données du cluster sur disque
- La définition de groupes composés chacun d'un serveur actif et un ou plusieurs des serveurs passifs qui permet de partitionner et répliquer des données (cette fonctionnalité n'est pas supportée dans la version open source)

La mise en oeuvre de ces fonctionnalités est effectuée dans le fichier de configuration.

L'état du cluster suite à l'arrêt du serveur actif (de façon volontaire ou non) dépend de deux facteurs :

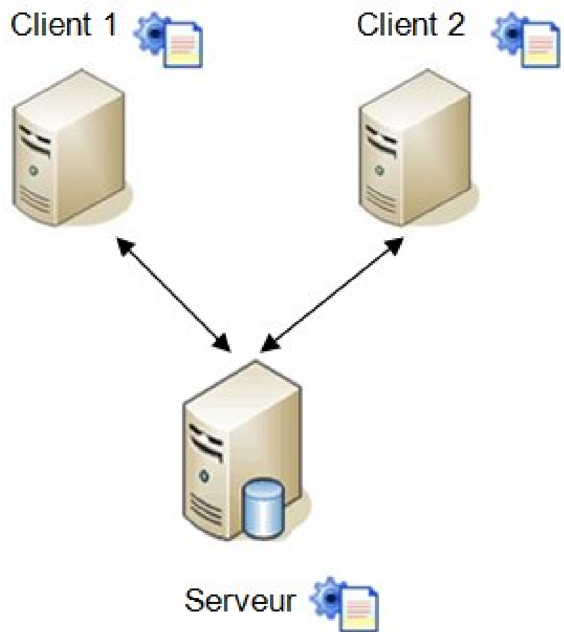
- Le serveur est dans le mode persistent ou non
- Une autre instance du serveur est configurée en mode passif

Serveur actif	Serveur passif	Etat du cluster
non persistent	non	Le cluster est inutilisable jusqu'au rédémarrage du serveur. A ce moment : • Les données des objets gérés sont perdues • Clients déjà connecté ne peuvent pas se reconnecter et doivent être arrêtés et redémarrés
non persistent	oui	le cluster continue de fonctionner car le serveur passif devient actif
persistent	non	Le cluster est inutilisable jusqu'au redémarrage du serveur. A ce moment : • Les données des objets gérés sont retrouvées • Clients déjà connecté peuvent se reconnecter
persistent	oui	le cluster continue de fonctionner car le serveur passif devient actif

47.8.1. La configuration minimale

Cette configuration ne propose ni fiabilité, ni haute disponibilité ni montée en charge.

Cette configuration est pratique dans un environnement de développement mais n'est absolument pas recommandée en production.



La configuration minimale du cluster doit utiliser :

- un seul serveur Terracotta
- en mode non persistant

Exemple :

```
<?xml version="1.0" encoding="UTF-8" ?>
<tc:tc-config xmlns:tc="http://www.terracotta.org/config"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.terracotta.org/schema/terracotta-5.xsd">
  <servers>
    <server name="noeud1">
...
          <dso>
              <persistence>
                  <mode>temporary-swap-only</mode>
              </persistence>
          </dso>
  </server>
...
  </servers>
  ...
</tc:tc-config>
```

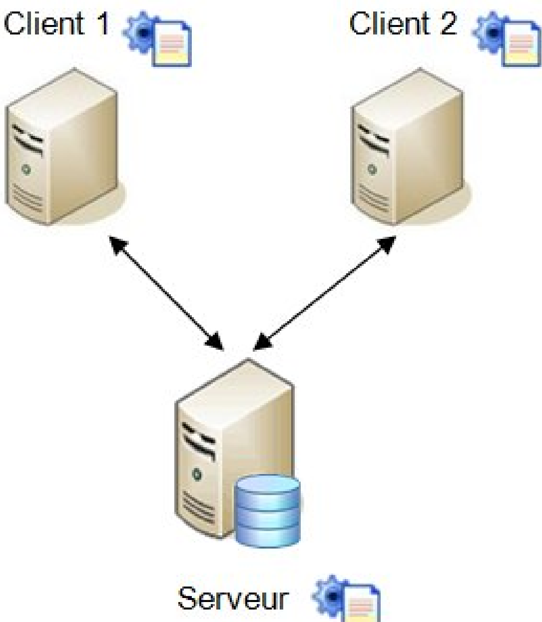
Pour démarrer le serveur, il suffit d'exécuter le script start-tc-server. L'option -f permet de préciser la localisation du fichier de configuration tc-config.xml.

47.8.2. La configuration pour la fiabilité

Cette configuration propose la fiabilité mais ne propose ni la haute disponibilité ni la montée en charge.

La configuration du cluster pour la fiabilité doit utiliser :

- Au moins un serveur Terracotta
- en mode persistant



Le mode persistant des données gérées par le cluster le rend plus fiable dans la mesure où les données sont restaurées si le cluster est redémarré.

Exemple :

```
<?xml version="1.0" encoding="UTF-8" ?>
<tc:tc-config xmlns:tc="http://www.terracotta.org/config"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.terracotta.org/schema/terracotta-5.xsd">
  <servers>
    <server name="noeud1">
...
          <dso>
              <persistence>
                  <mode>permanent-store</mode>
              </persistence>
          </dso>
  </server>
...
  </servers>
  ...
</tc:tc-config>
```

Cette configuration n'est généralement pas souhaitable telle quelle ni dans un environnement de développement (la persistance données n'est pas nécessaires et parfois peu pratique car la purge des données est manuelle), ni dans un environnement de production (car elle n'assure pas la haute disponibilité).

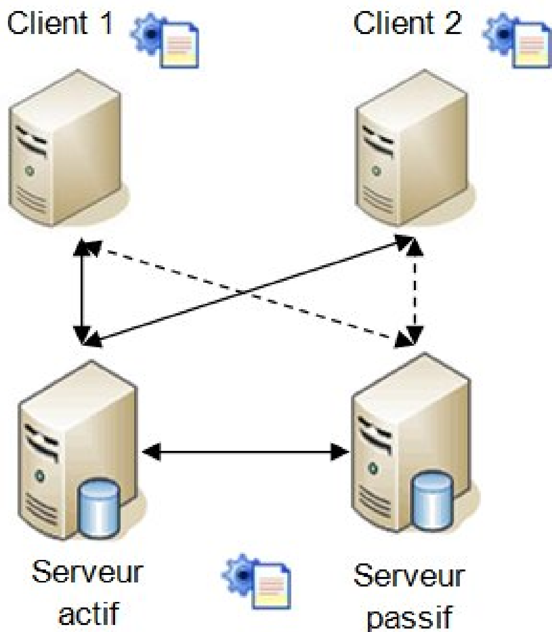
47.8.3. La configuration pour une haute disponibilité

En production, il est nécessaire d'assurer la haute disponibilité du cluster notamment en mettant en place un failover sur le serveur.

La configuration du cluster pour une haute disponibilité doit utiliser :

- au moins deux serveurs Terracotta
- le mode actif-passif

Dans cette configuration, un serveur actif communique avec les clients du cluster. Au moins un serveur passif attend de prendre le relai pour devenir le serveur actif en cas de défaillance du serveur actif. Dans cette configuration, un seul serveur peut être actif.



Terracotta synchronise automatiquement les serveurs pour permettre aux serveurs passifs d'être dans le même état que le serveur actif et ainsi pouvoir prendre sa place en cas de défaillance.

Si les serveurs sont démarrés simultanément, un est choisi pour être le serveur actif, les autres ont le rôle passif. Lors de démarrage d'un serveur, si un serveur actif est en cours d'exécution, son état est synchronisé avec le serveur démarré.

En cas d'arrêt du serveur actif, un des serveurs passif est promu actif.

Le mode actif-passif est configuré dans le tag <mode> fils du tag servers/ha avec la valeur networked-active-passive qui est celle recommandée car elle assure la synchronisation via le réseau.

Exemple :

```
<?xml version="1.0" encoding="UTF-8" ?>
<tc:tc-config xmlns:tc="http://www.terracotta.org/config"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.terracotta.org/schema/terracotta-5.xsd">
  <servers>
    <server host="host2" name="noeud1">
...
    </server>
    <server host="host1" name="noeud2">
...
    </server>
...
     <ha>
        <mode>networked-active-passive</mode>
           <networked-active-passive>
               <election-time>5</election-time>
           </networked-active-passive>
       </ha>
  </servers>
  ...
</tc:tc-config>
```

Chaque serveur doit être définit dans un tag <server> avec obligatoirement pour chacun un attribut name unique.

Le tag <election-time> permet de préciser une durée en seconde pour déterminer le nouveau serveur actif. La valeur par défaut est 5 secondes.

Dans cette configuration, il est important que les répertoires de données précisés dans le tag <data> de chaque serveur soient différents et de préférence sur la machine locale pour améliorer les performances lorsque la persistance est requise.

Les répertoires des tags <logs> et <statistics> doivent aussi être différents.

Pour démarrer le serveur, il suffit d'exécuter le script start-tc-server avec l'option -n suivie du nom du serveur à démarrer. L'option -f permet de préciser la localisation du fichier de configuration tc-config.xml.

47.8.4. La configuration pour une haute disponibilité et la fiabilité

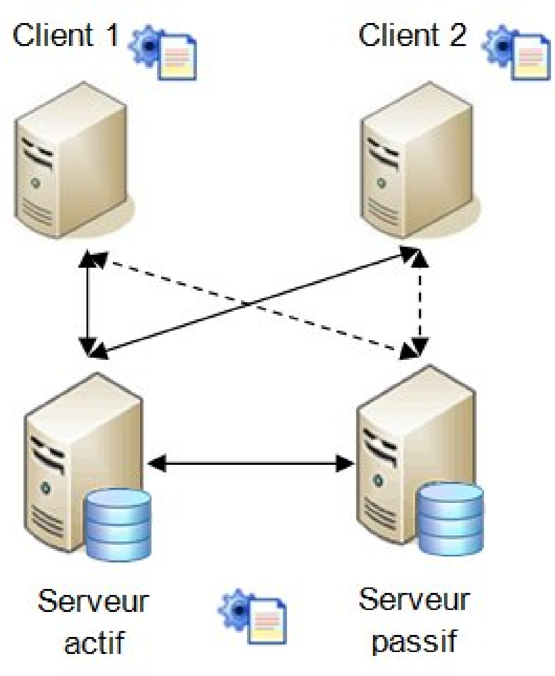
Cette configuration propose la fiabilité et la haute disponibilité.

En production, il est nécessaire d'assurer la haute disponibilité et la fiabilité du cluster notamment en mettant en place un failover sur le serveur et la persistance des données.

La configuration du cluster pour une haute disponibilité et la fiabilité doit utiliser :

- au moins deux serveurs Terracotta
- en mode persistance des données
- le mode actif-passif

Cette configuration est une combinaison des configurations haute disponibilité et fiabilité.



Exemple :

```
<?xml version="1.0" encoding="UTF-8" ?>
<tc:tc-config xmlns:tc="http://www.terracotta.org/config"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.terracotta.org/schema/terracotta-5.xsd">
  <servers>
    <server host="host2" name="noeud1">
...
          <data>repertoire_partage_des_donnees</data>
          <dso>
              <persistence>
                  <mode>permanent-store</mode>
              </persistence>
          </dso>
    </server>
    <server host="host1" name="noeud2">
...
          <data>repertoire_partage_des_donnees</data>
          <dso>
              <persistence>
                  <mode>permanent-store</mode>
              </persistence>
          </dso>
    </server>
...
     <ha>
        <mode>networked-active-passive</mode>
           <networked-active-passive>
               <election-time>5</election-time>
           </networked-active-passive>
       </ha>
  </servers>
  ...
</tc:tc-config>
```

Il est possible d'utiliser plus de deux serveurs, un étant le serveur actif qui communique avec les clients, les autres serveurs étant passif.

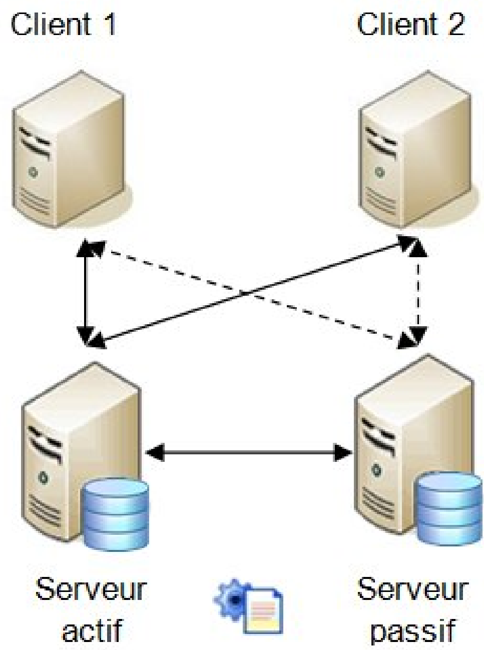
Les serveurs passifs peuvent être configurés en mode persistant ou non persistant mais il est préférable de les configurer dans le même mode que le serveur actif.

Si le actif en mode persistant s'arrête et que le nouveau serveur actif est non persistant, il faudra obligatoirement purger les données sur disque avant de relancer le serveur, car le serveur actif courant ne persiste plus les données sur disque et

il risque d'y avoir une incohérence entre les données en mémoire et celles sur disque.

47.8.5. La configuration pour un environnement de production

En production, il est préférable de n'avoir qu'un seul fichier de configuration pour tous les éléments du cluster. Ceci n'est pas obligatoire mais cela facilite la gestion et la maintenance de centraliser le fichier de configuration à un seul endroit plutôt que de le répliquer pour chaque élément du cluster.



Le fichier de configuration est accessible aux différents serveurs en étant stockée dans un répertoire partagé. Lorsque le client se connecte à un serveur, il lui demande le contenu du fichier de configuration.

Dans le fichier de configuration, il faut que chaque serveur soit précisément identifié par leur host et par un nom unique.

Terracotta propose que les clients obtiennent le fichier de configuration de la part du serveur auquel ils sont connectés. Pour cela, il faut fournir le host suivi de deux points suivi du port du serveur à la place du chemin du fichier de configuration dans la variable d'environnement TC_CONFIG_PATH. Il est possible de préciser les serveurs de cluster en les séparant par une virgule.

47.8.6. La configuration pour la montée en charge

Cette configuration propose la fiabilité, la haute disponibilité et la montée en charge.

La montée en charge est proposée au travers de la fonctionnalité mirror groups qui permet de définir des groupes de serveurs. Chacun de ces groupes possède un serveur actif et un ou plusieurs serveurs passifs.

Cette fonctionnalité n'est pas prise charge dans la version open source de Terracotta mais elle est disponible dans la version enterprise.

47.9. Quelques recommandations

Il est important pour le développeur de garder à l'esprit que pour que le cluster mis en oeuvre avec Terracotta fonctionne, l'application doit être codée en gérant correctement les accès concurrents.

Il est aussi nécessaire de prendre plusieurs facteurs en compte :

- s'assurer que tous les types d'objets gérés par Terracotta sont instrumentés dans le fichier de configuration
- correctement configurer les différents éléments dans le fichier de configuration
- ...

En cas de problème, Terracotta est assez verbeux dans l'exception levée et propose même une ou plusieurs pistes de corrections qui sont assez pertinentes.

Partie 7 : Développement d'applications d'entreprises

Cette quatrième partie traite d'une utilisation de Java en forte expansion : le développement côté serveur. Ce type de développement est poussé par l'utilisation d'Internet notamment.

Ces développements sont tellement importants que Sun propose une véritable plate-forme, basée sur le J2SE et orientée entreprise dont les API assurent le développement côté serveur : J2EE (Java 2 Entreprise Edition).

Cette partie regroupe plusieurs chapitres :

- J2EE / Java EE : introduit la plate-forme Java 2 Entreprise Edition
- JavaMail : traite de l'API qui permet l'envoie et la réception d'e mail
- JMS (Java Messaging Service) : indique comment utiliser cette API qui permet l'utilisation de système de messages pour l'échange de données entre applications
- Les EJB (Entreprise Java Bean) : propose une présentation de l'API et les spécifications pour des objets chargés de contenir les règles métiers
- Les EJB 3 : ce chapitre détaille la version 3 des EJB qui est une évolution majeure de cette technologie car elle met l'accent sur la facilité de développement sans sacrifier les fonctionnalités qui font la force des EJB.
- Les EJB 3.1 : ce chapitre détaille la version 3.1 des EJB utilisée par Java EE 6
- Les services web de type Soap : permettent l'appel de services distants en utilisant un protocole de communication et une structuration des données échangées avec XML de façon standardisée

48. J2EE / Java EE

J2EE est l'acronyme de Java 2 Entreprise Edition. Cette édition est dédiée à la réalisation d'applications pour entreprises. J2EE est basé sur J2SE (Java 2 Standard Edition) qui contient les API de base de Java. Depuis sa version 5, J2EE est renommé Java EE (Enterprise Edition).

Ce chapitre contient plusieurs sections :

- La présentation de J2EE : présente rapidement la plate-forme J2EE
- Les API de J2EE : présente rapidement les différentes API qui composent J2EE
- L'environnement d'exécution des applications J2EE : présente les différents éléments qui compose l'environnement d'exécution des applications J2EE
- L'assemblage et le déploiement d'applications J2EE : décrit le mode d'assemblage et de déploiement des applications J2EE
- J2EE 1.4 SDK : installation et prise en main du J2EE 1.4 SDK
- La présentation de Java EE 5.0 : présente rapidement la versoin 5 plate-forme Java EE
- La présentation de Java EE 6

48.1. La présentation de J2EE

J2EE est une plate-forme fortement orientée serveur pour le développement et l'exécution d'applications distribuées. Elle est composée de deux parties essentielles :

- un ensemble de spécifications pour une infrastructure dans laquelle s'exécutent les composants écrits en Java : un tel environnement se nomme serveur d'application.
- un ensemble d'API qui peut être obtenu et utilisé séparément. Pour être utilisées, certaines nécessitent une implémentation de la part d'un fournisseur tiers.

Sun propose une implémentation minimale des spécifications de J2EE : le J2EE SDK. Cette implémentation permet de développer des applications respectant les spécifications mais n'est pas prévue pour être utilisée dans un environnement de production. Ces spécifications doivent être respectées par les outils développés par des éditeurs tiers.

L'utilisation de J2EE pour développer et exécuter une application propose plusieurs avantages :

- une architecture d'application basée sur les composants qui permet un découpage de l'application et donc une séparation des rôles lors du développement
- la possibilité de s'interfacer avec le système d'information existant grâce à de nombreuses API : JDBC, JNDI, JMS, JCA ...
- la possibilité de choisir les outils de développement et le ou les serveurs d'applications utilisés qu'ils soient commerciaux ou libres

J2EE permet une grande flexibilité dans le choix de l'architecture de l'application en combinant les différents composants. Ce choix dépend des besoins auxquels doit répondre l'application mais aussi des compétences dans les différentes API de J2EE. L'architecture d'une application se découpe idéalement en au moins trois tiers :

- la partie cliente : c'est la partie qui permet le dialogue avec l'utilisateur. Elle peut être composée d'une application standalone, d'une application web ou d'applets

* la partie métier : c'est la partie qui encapsule les traitements (dans des EJB ou des JavaBeans)
* la partie données : c'est la partie qui stocke les données

La suite de ce chapitre sera développée dans une version future de ce document

48.2. Les API de J2EE

J2EE regroupe un ensemble d'API pour le développement d'applications d'entreprise.

API	Rôle	version de l'API dans		
		J2EE 1.2	J2EE 1.3	J2EE 1.4
Entreprise Java Bean (EJB)	Composants serveurs contenant la logique métier	1.1	2.0	2.1
Remote Method Invocation (RMI) et RMI-IIOP	RMI permet l'utilisation d'objets Java distribué. RMI-IIOP est une extension de RMI pour une utilisation avec CORBA.	1.0		
Java Naming and Directory Interface (JNDI)	Accès aux services de nommage et aux annuaires d'entreprises	1.2	1.2	1.2.1
Java Database Connectivity (JDBC)	Accès aux bases de données. J2EE intègre une extension de cette API	2.0	2.0	3.0
Java Transaction API (JTA) Java Transaction Service (JTS)	Support des transactions	1.0	1.0	1.0
Java Messaging service (JMS)	Support de messages via des MOM (Messages Oriented Middleware)	1.0	1.0	1.1
Servlets	Composants basés sur le concept C/S pour ajouter des fonctionnalités à un serveur. Pour le moment, principalement utilisé pour étendre un serveur web	2.2	2.3	2.4
Java Server Pages (JSP)		1.1	1.2	2.0
Java IDL	Utilisation de CORBA			
JavaMail	Envoie et réception d'email	1.1	1.2	1.3
J2EE Connector Architecture (JCA)	Connecteurs pour accéder à des ressources du système d'information de l'entreprises telles que CICS, TUXEDO, SAP ...		1.0	1.5
Java API for XML Parsing (JAXP)	Analyse et exploitation de données au format XML		1.1	1.2
Java Authentication and Authorization Service (JAAS)	Echange sécurisé de données		1.0	
JavaBeans Activation Framework	Utilisé par JavaMail : permet de déterminer le type mime		1.0.2	1.0.2
Java API for XML-based RPC (JAXP-RPC)				1.1
				1.2

SOAP with Attachments API for Java (SAAJ)				
Java API for XML Registries (JAXR)				1.0
Java Management Extensions (JMX)				1.2
Java Authorization Service Provider Contract for Containers (JACC)				1.0

Ces API peuvent être regroupées en trois grandes catégories :

- les composants : Servlet, JSP, EJB
- les services : JDBC, JTA/JTS, JNDI, JCA, JAAS
- la communication : RMI-IIOP, JMS, Java Mail

48.3. L'environnement d'exécution des applications J2EE

J2EE propose des spécifications pour une infrastructure dans laquelle s'exécutent les composants. Ces spécifications décrivent les rôles de chaque élément et précisent un ensemble d'interfaces pour permettre à chacun de ces éléments de communiquer.

Ceci permet de séparer les applications et l'environnement dans lequel il s'exécute. Les spécifications précisent à l'aide des API un certain nombre de fonctionnalités que doivent implémenter l'environnement d'exécution. Ces fonctionnalités sont de bas niveau ce qui permet aux développeurs de se concentrer sur la logique métier.

Pour exécuter ces composants de natures différentes, J2EE défini des conteneurs pour chacun de ces composants. Il définit pour chaque composant des interfaces qui leur permettront de dialoguer avec les composants lors de leur exécution. Les conteneurs permettent aux applications d'accéder aux ressources et aux services en utilisant les API.

Les appels aux composants se font par des clients via les conteneurs. Les clients n'accèdent pas directement aux composants mais sollicitent le conteneur pour les utiliser.

48.3.1. Les conteneurs

Les conteneurs assurent la gestion du cycle de vie des composants qui s'exécutent en eux. Les conteneurs fournissent des services qui peuvent être utilisés par les applications lors de leur exécution.

Il existe plusieurs conteneurs définit par J2EE:

- conteneur web : pour exécuter les servlets et les JSP
- conteneur d'EJB : pour exécuter les EJB
- conteneur client : pour exécuter des applications standalone sur les postes qui utilisent des composants J2EE

Les serveurs d'applications peuvent fournir un conteneur web uniquement (exemple : Tomcat) ou un conteneur d'EJB uniquement (exemple : JBoss, Jonas, ...) ou les deux (exemple : Websphere, Weblogic, ...).

Pour déployer une application dans un conteneur, il faut lui fournir deux éléments :

- l'application avec tous les composants (classes compilées, ressources ...) regroupée dans une archive ou module. Chaque conteneur possède son propre format d'archive.
- un fichier descripteur de déploiement contenu dans le module qui précise au conteneur des options pour exécuter l'application

Il existe trois types d'archives :

Archive / module	Contenu	Extension	Descripteur de déploiement
bibliothèque	Regroupe des classes	jar	
application client	Regroupe les ressources nécessaires à leur exécution (classes, bibliothèques, images, ...)	jar	application-client.jar
web	Regroupe les servlets et les JSP ainsi que les ressources nécessaires à leur exécution (classes, bibliothèques de balises, images, ...)	war	web.xml
EJB	Regroupe les EJB et leurs composants (classes)	jar	ejb-jar.xml

Une application est un regroupement d'un ou plusieurs modules dans un fichier EAR (Entreprise ARchive). L'application est décrite dans un fichier application.xml lui même contenu dans le fichier EAR

48.3.2. Le conteneur web

Le conteneur web est une implémentation des spécifications servlets et par extension des spécifications des JSP. Ce type de conteneur est composé de deux éléments majeur : un moteur de servlets (servlets engine) et un moteur de JSP (JSP engine).

Les conteneurs web peuvent généralement utiliser leur propre serveur web et être utilisés en tant que plug in d'un serveur web dédié (Apache, IIS, ...).

L'implémentation de référence pour ce type de conteneur est le projet open source Tomcat du groupe Apache.

Les API spécifiquement mises en oeuvre dans un conteur web sont détaillées dans les chapitres «Les servlets»" et "JSP".

48.3.3. Le conteneur d'EJB

Les EJB sont détaillées dans le chapitre «Les EJB (Entreprise Java Bean)».

48.3.4. Les services proposés par la plate-forme J2EE

Une plate-forme d'exécution J2EE complète implémentée dans un serveur d'application propose les services suivants :

- service de nommage (naming service)
- service de déploiement (deployment service)
- service de gestion des transactions (transaction service)
- service de sécurité (security service)

Ces services sont utilisés directement ou indirectement par les conteneurs mais aussi par les composants qui s'exécutent dans les conteneurs grâce à leurs API respectives.

48.4. L'assemblage et le déploiement d'applications J2EE

J2EE propose une spécification pour décrire le mode d'assemblage et de déploiement d'une application J2EE.

Une application J2EE peut regrouper différents modules : modules web, modules EJB ... Chacun de ces modules possède son propre mode de packaging. J2EE propose de regrouper ces différents modules dans un module unique sous la forme d'un fichier EAR (Entreprise ARchive).

Le format de cette archive est très semblable à celui des autres archives :

- un contenu : les différents modules qui composent l'application (module web, EJB, fichier RAR, ...)
- un fichier descripteur de déploiement

Les serveurs d'application extraient chaque module du fichier EAR et les déploient séparément un par un.

48.4.1. Le contenu et l'organisation d'un fichier EAR

Le fichier EAR est composé au minimum :

- d'un ou plusieurs modules
- d'un répertoire META-INF contenant un fichier descripteur de déploiement nommé application.xml

Les modules ne doivent pas obligatoirement être insérés à la racine du fichier EAR : ils peuvent être mis dans un des sous répertoires pour organiser le contenu de l'application. Il est par exemple pratique de créer un répertoire lib qui contient les fichiers .jar des bibliothèques communes aux différents modules.

48.4.2. La création d'un fichier EAR

Pour créer un fichier EAR, il est possible d'utiliser un outil graphique fourni par le vendeur du serveur d'application ou de créer le fichier manuellement en suivant les étapes suivantes :

1. créer l'arborescence des répertoires qui vont contenir les modules
2. insérer dans cette arborescence les différents modules à inclure dans le fichier EAR
3. créer le répertoire META-INF (en respectant la casse)
4. créer le fichier application.xml dans ce répertoire
5. utiliser l'outil jar pour créer le fichier EAR en précisant les options cvf, le nom du fichier ear avec son extension et les différents éléments qui composent le fichier (modules, répertoire dont le répertoire META-INF).

48.4.3. Les limitations des fichiers EAR

Actuellement les fichiers EAR ne servent qu'à regrouper différents modules pour former une seule entité. Rien n'est actuellement prévu pour prendre en compte la configuration des objets permettant l'accès aux ressources par l'application telles qu'une base de données (JDBC pour DataSource, pool de connexion ...), un système de message (JMS), etc ...

Pour lever une partie de ces limites, les serveurs d'applications commerciaux proposent souvent des mécanismes propriétaires supplémentaires pour palier à ces manques en attendant une évolution des spécifications.

48.5. J2EE 1.4 SDK

La version 1.4 de J2EE a été diffusée en novembre 2003.

La grande nouveauté de la version 1.4 est le support des services web. Deux nouvelles API ont été ajoutées pour

normaliser le déploiement (J2EE deployment API 1.1) et la gestion des applications (J2EE management API 1.0 qui utilise JMX). Une nouvelle API permet de standardiser l'authentification (Java ACC : Java Authorization Contract for Container). Plusieurs API déjà présentes dans les précédentes versions de J2EE ont été mises à jour (EJB, JSP, Servlet, ...) :

- J2EE Connector Architecture 1.5
- Enterprise JavaBeans (EJB) 2.1
- JavaServer Pages (JSP) 2.0
- Java Servlet 2.4
- JavaMail 1.3
- Java Message Service 1.1
- Java API for XML parsing (JAXP) 1.2
- Java API for XML-based RPC (JAX-RPC) 1.1
- SOAP with Attachments API for Java (SAAJ) 1.2
- Java API for XML Registries (JAXR) 1.0
- Java Management Extensions (JMX) 1.2
- Java Authorization Service Provider Contract for Containers (JACC) 1.0

Le J2EE SDK 1.4 qui est l'implémentation de référence inclus le J2SE SDK 1.4.2 et J2EE 1.4 application server.

48.5.1. L'installation de l'implémentation de référence sous Windows

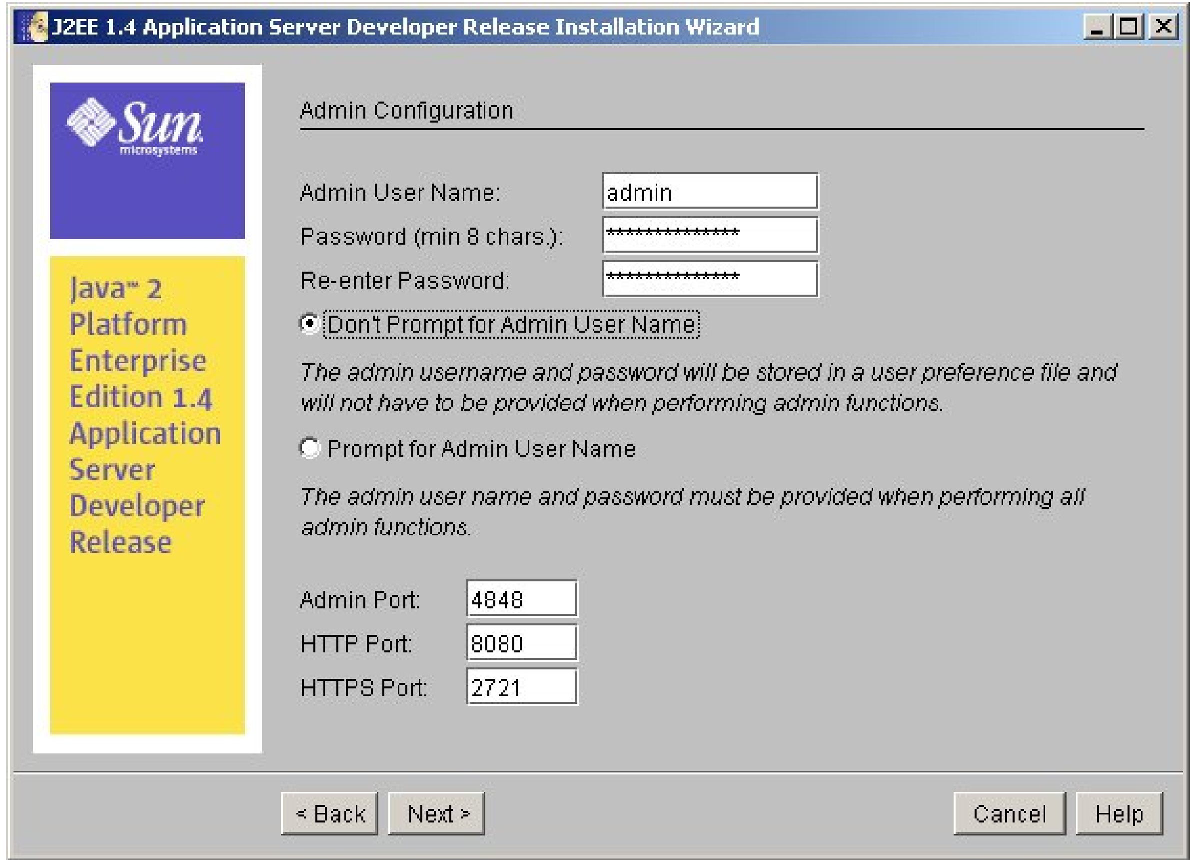
Le J2EE SDK 1.4 peut être installé sur les systèmes Microsoft suivants : Windows 2000 pro avec un service pack SP2, Windows XP PRO avec un service pack SP1 et Windows Server 2003.

Il existe plusieurs packages d'installation : celui utilisé ci dessous ne contient que le serveur d'application puisque le J2SE 1.4.2 était déjà présent sur la machine.

Lancer le programme j2eesdk-1_4-dr-windows-eval-app.exe. L'application extrait les fichiers, lance le J2RE et exécute un assistant qui va guider l'installation :

- la première page est la page d'accueil (welcome) : cliquez sur le bouton « Next »
- La page suivante permet de lire et d'accepter la licence d'utilisation (software licence agreement) : lire la licence et si vous l'acceptez, cliquez sur le bouton « Yes » puis sur le bouton « Next »
- la page suivante permet de sélectionner le répertoire d'installation du produit (Select Installation Directory) : sélectionnez le répertoire d'installation et cliquez sur « Next »

  Cliquez sur « Create directory »
- la page suivante permet de préciser l'emplacement du J2SDK nécessaire au produit (Java 2 SDK Required) : sélectionnez l'emplacement du J2SE SDK (si il est présent sur la machine, son chemin est proposé par défaut), puis cliquez sur le bouton « Next ».
- la page suivante permet de préciser les informations nécessaires la configuration du serveur



Il faut saisir des paramètres de configuration : saisir le mot de passe de l'administrateur et sélectionner l'option pour l'authentification ou non lors d'actions d'administration

Il est aussi possible de définir les ports pour le module d'administration et pour les serveurs web HTTP et HTTPS.

Une fois les informations saisies, cliquez sur le bouton « Next ».

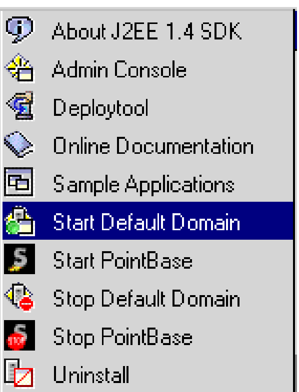
- La page suivante permet de sélectionner le type d'installation à réaliser (Installation Options) : pour une première installation, il suffit de conserver l'option par défaut proposée puis cliquez sur le bouton « Next »
- La page suivante synthétise les différentes informations avant l'installation (Ready to Install) : cliquez sur « Install Now »
- La dernière page indique que l'installation s'est bien terminée et donne quelques informations sommaires sur les différents outils

Il est utile de rajouter le répertoire bin du répertoire d'installation de J2EE SDK 1.4 à la variable PATH du système d'exploitation.

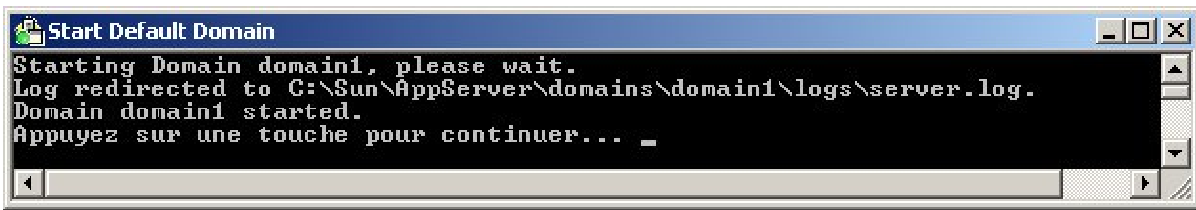
48.5.2. Le démarrage et l'arrêt du serveur

Un domaine permet de regrouper des applications avec une configuration particulière qui s'exécutent sur une instance particulière du serveur. Lors de l'installation un domaine par défaut est créé : domain1

Le programme d'installation a créé plusieurs entrées dans le menu « Démarrer/Programmes/Sun Microsystems/J2EE 1.4 SDK/ »

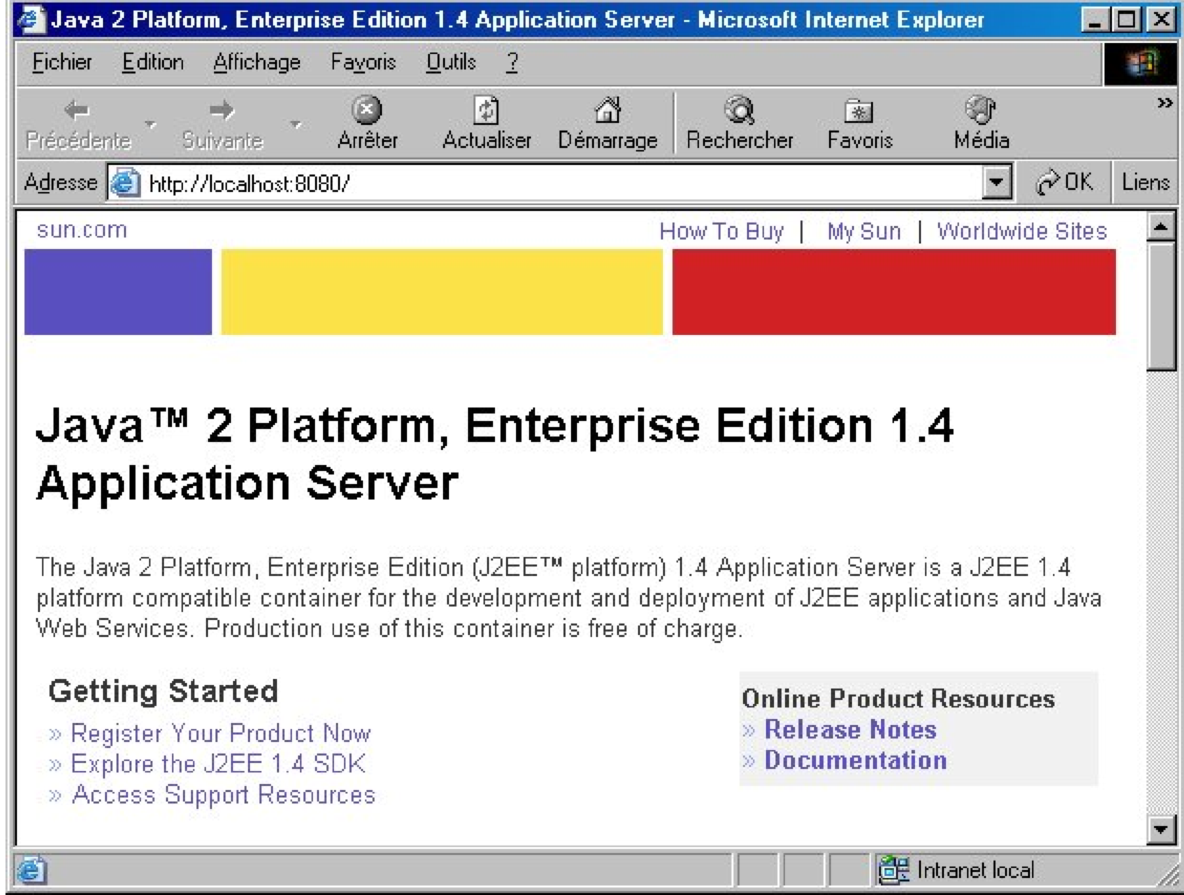


L'option « Start default domain » permet de démarrer le domaine domain1.



Il suffit d'appuyer sur une touche pour fermer la fenêtre.

Pour vérifier la bonne exécution du serveur, il suffit d'appeler l'URL http://localhost:nnnn/ dans un navigateur où nnnn représente le port http précisé dans les paramètres lors de l'installation.



L'arrêt du domaine par défaut peut être obtenu en utilisant l'option « Stop default domain ».

48.5.3. L'outil asadmin

J2EE application server est livré avec une application nommée asadmin, utilisable sur une ligne de commandes, pour administrer le serveur.

Cette application utilise deux modes de fonctionnement :

- la réception de commandes par la console après son lancement
- le passage des commandes en argument de la commande

Les commandes possèdent des noms bien définis en fonction de leurs actions et nécessite souvent un ou plusieurs paramètres.

Exemple : démarrage d'un domaine

C:\>asadmin start-domain domain1

Starting Domain domain1, please wait.

Log redirected to C:\Sun\AppServer\domains\domain1\logs\server.log.

Domain domain1 started.

Exemple : arrêt d'un domaine

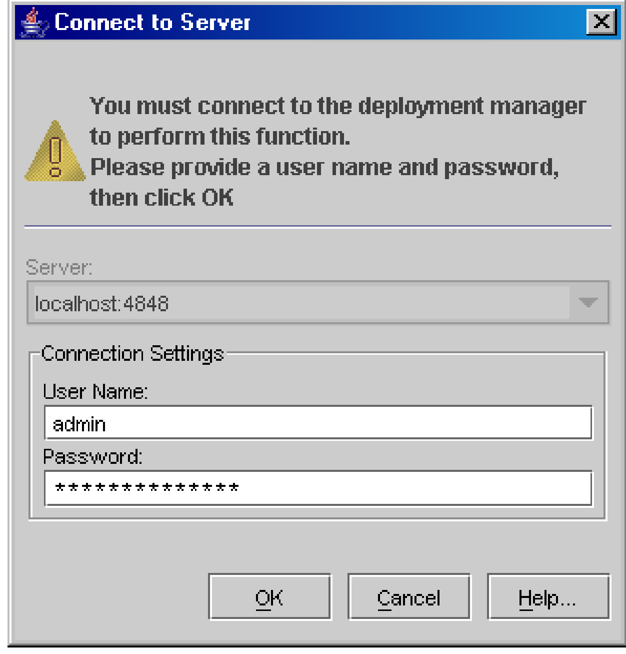
C:\>asadmin stop-domain domain1

Domain domain1 stopped.

48.5.4. Le déploiement d'applications

Pour déployer une application sous la forme d'un fichier war ou ear il suffit de copier le fichier dans le sous répertoire domains/nom_du_domaine/autodeploy.

Il est nécessaire de s'authentifier auprès du serveur d'application pour certaines opérations.



48.5.5. La console d'administration

La console d'administration est une application web qui permet de configurer le serveur.

Pour l'utiliser, le serveur doit être lancé et il suffit de saisir dans un navigateur l'Url http://localhost:4848/asadmin

48.6. La présentation de Java EE 5.0

Le nom de la cinquième version de la plate-forme Java pour Entreprise a été simplifiée : au lieu de se nommer J2EE (Java 2 Enterprise Edition) version 1.5, la plate-forme a été renommée Java EE 5 (Java Enterprise Edition).

L'accent est mis dans cette version sur la simplification des développements tout en conservant et en faisant évoluer les fonctionnalités proposées par la plate-forme J2EE.

J2EE est réputé pour sa complexité et pour certaines lourdeurs essentiellement liés aux nombreuses entités à développer (classes, interfaces, fichiers de configuration, ...). La version 5 de la plate-forme repose sur la version 5 de la plate-forme Java Standard Edition et profite donc de ces améliorations notamment les generics et les annotations. L'utilisation intensive de ces dernières dans la version 5 de la plate-forme Java EE permet de simplifier les développements et ainsi de réduire le temps nécessaire à leur réalisation.

L'un des principaux buts de Java EE 5 est de conserver les fonctionnalités et la puissance de la plate-forme tout en simplifiant grandement le code à produire. Cette simplification repose d'une façon globale essentiellement sur :

- L'utilisation intensive des annotations afin de réduire le volume de code et le nombre de fichiers à créer
- L'utilisation de valeurs et de comportements par défaut
- Les descripteurs de déploiement ne sont plus nécessaires que pour des besoins très particuliers

La simplification concerne aussi des sujets plus précis tel que l'utilisation des POJO ou de l'injection de ressources.

Cette nouvelle version de la plate-forme, spécifiée dans la JSR 244, propose donc d'énormes simplifications dans le code à écrire par les développeurs.

Elle intègre aussi de nouvelles API :

- Java Server Faces pour le développement d'applications web avec la JSTL et un support pour AJAX

* Une nouvelle API pour la persistance des données reposant sur les POJO et les annotations : Java Persistence API
* La version 3.0 des EJB simplifie grandement l'utilisation de cette technologie
* Le support des dernières versions des API concernant les services web permettant une mise en oeuvre d'une architecture de type SOA

Elle intègre aussi les dernières versions de la plupart des API qui formaient la version précédente de la plate-forme. Ainsi la version 5 de l'édition entreprise de Java inclue de nombreuses spécifications :

Technologies pour les services web	
Implementing Enterprise Web Services	JSR 109
Java API for XML-Based Web Services (JAX-WS) 2.0	JSR 224
Java API for XML-Based RPC (JAX-RPC) 1.1	JSR 101
Java Architecture for XML Binding (JAXB) 2.0	JSR 222
SOAP with Attachments API for Java (SAAJ)	JSR 67
Web Service Metadata for the Java Platform	JSR 181
Technologies pour les composants	
Enterprise JavaBeans 3.0	JSR 220
J2EE Connector Architecture 1.5	JSR 112
Java Servlet 2.5	JSR 154
JavaServer Faces 1.2	JSR 252
JavaServer Pages 2.1	JSR 245
JavaServer Pages Standard Tag Library	JSR 52
Technologies de gestion et déploiement	
J2EE Management	JSR 77
J2EE Application Deployment	JSR 88
Java Authorization Contract for Containers	JSR 115
Autres technologies	
Common Annotations for the Java Platform	JSR 250
Java Transaction API (JTA)	JSR 907
JavaBeans Activation Framework (JAF) 1.1	JSR 925
JavaMail	JSR 919
Streaming API for XML (StAX) 1.0	JSR 173

48.6.1. La simplification des développements

La version 5 de la plate-forme Java EE fait un usage important des annotations introduites dans la plate-forme Java SE 5.0. Les annotations sont des méta-datas qui seront utilisées par le conteneur. Le code à écrire est ainsi réduit car certaines entités à définir ou règles à respecter sont simplement remplacées par l'utilisation d'une ou plusieurs annotations.

Historiquement, des tags étaient déjà utilisés notamment par Javadoc.

Une annotation commence par le caractère @ suivi par le nom de l'annotation éventuellement suivi d'une liste, entourée de parenthèses, de paramètres sous la forme de paire clé/valeur.

Les annotations précèdent par convention les modificateurs des entités qu'elles caractérisent. Elles correspondent à des classes particulières.

Les annotations n'ont aucune influence sur la logique des traitements du code mais elles influent sur la façon dont certains outils vont exécuter le code.

Java EE 5 propose des annotations pour de nombreux rôles :

- définition et utilisation des services web
- définition des EJB
- mapping objets / XML
- mapping objet / relationnel
- précision sur les informations de déploiement
- ...

L'utilisation des annotations n'est pas obligatoire : la configuration peut aussi être faite dans un descripteur de déploiement.

Le packaging a aussi été simplifié : il est maintenant possible d'écrire des EJB ou des services web sans devoir écrire de descripteur de déploiement sauf pour des besoins particuliers. La configuration est déduite par le conteneur à partir des annotations.

De nombreux attributs d'annotations possèdent des valeurs par défaut, ce qui évite au développeur de devoir les préciser dans le cas ou cette valeur par défaut est celle à utiliser.

48.6.2. La version 3.0 des EJB

La version 3.0 des EJB propose une simplification de leur développement. Le travail des développeurs est réduit au profit d'une augmentation des traitements pris en charge par le conteneur :

- Le nombre de classes et d'interfaces à écrire est réduit
- Le descripteur de déploiement est optionnel car il n'est plus utile que pour besoin spécifique. Pour cela, la définition des composants utilise les annotations et l'injection de dépendance.
- Les EJB entités sont plus facile à développer en faisant usage de la nouvelle API Java Persistence API pour le mapping O/R
- Les intercepteurs permettent de proposer des traitements avant ou après l'appel de méthodes à l'image de ce que peuvent proposer certaines fonctionnalités de l'AOP

Dans les versions antérieures des spécifications, les interactions entre le bean et le conteneur pour la gestion de son cycle de vie étaient réalisées via les méthodes ejbRemove(), setMessage(), setSessionContext(), ejbActivate(), et ejbPassivate() des classes javax.ejb.SessionBean and javax.ejb.MessageDrivenBean. Même inutiles, ces méthodes devaient être écrites.

Dans la version 3.0, il suffit simplement d'utiliser les annotations définies dans les spécifications de Java EE dont voici les principales :

tag	Rôle
@Stateless	annote une classe qui est un composant de type EJB session stateless
@Stateful	annote une classe qui est un composant de type EJB session stateful
@PostConstruct	
@PreDestroy	
@PostActivate	
@PrePassivate	
@EJB	annote un EJB qui sera injectée par le conteneur
@WebServiceRef	annote un service web qui sera injecté par le conteneur

@Resource	annote une ressource différente d'un EJB ou d'un service web qui sera injectée par le conteneur
@MessageDriven	annote une classe qui est un composant de type EJB Message Driven
@TransactionAttribute	annote une classe (dans ce cas toutes ses méthodes) ou une méthode pour préciser les attributs d'appartenance à une transaction
@TransactionManagement	
@RolesAllowed, @PermitAll @DenyAll	annote une méthode pour indiquer ses permissions d'utilisation
@RolesReferenced	
@RunAs	

48.6.3. Un accès facilité aux ressources grâce à l'injection de dépendance

Le motif de conception injection de dépendance permet à une entité extérieure à un objet de lui fournir toutes les références sur les objets dont il dépend.

Avec Java EE 5, l'injection de dépendance peut être utilisée sur plusieurs types de ressources utilisées par un composant :

- EJB
- Services web
- DataSource
- EntityManager
- Queue et Topic
- SessionContext
- UserTransaction
- TimerService

Trois annotations permettent de mettre en oeuvre l'injection de dépendance :

- @EJB : pour les EJB
- @WebServiceRef : pour les services web
- @Ressource : pour tous les autres types de ressources supportées

Ces annotations peuvent être utilisées dans tout objet dont le cycle de vie est géré par un conteneur du serveur d'application (EJB, service web, servlet, bean entité, ...).

L'injection de dépendance peut donc être mise en oeuvre dans les trois conteneurs de la plate-forme : EJB, web et client.

Ceci permet d'éviter l'utilisation directe de l'API JNDI pour obtenir une instance de la ressource stockée dans l'annuaire.

48.6.4. JPA : le nouvelle API de persistance

La nouvelle API Java Persistence API, développée sous la JSR 220, est ajoutée à la version 5.0 de la plate-forme Java EE. Cette API peut être utilisée dans les EJB mais aussi dans toute autre application même celle utilisant Java SE. Son utilisation n'est ainsi pas réservée qu'à des développements avec la plate-forme Java EE.

Cette API propose les fonctionnalités suivantes :

- standardisation du mapping O/R
- utilisation de POJO
- mise en oeuvre des opérations de type CRUD au travers de l'objet EntityManager
- support de l'héritage et du polymorphisme

* requêtes en EJB Query Langage

Les entités sont de simples POJO enrichis d'annotations dédiées. Ces annotations permettent de préciser comment est réalisé le mapping entre une ou plusieurs tables d'une base de données et l'entité

Exemple :

```
package com.jmdoudoux.test.jpa;

import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@Entity
public class Personne implements Serializable {
  @Id
  @GeneratedValue
  private int id;

  private String prenom;

  private String nom;

  private static final long serialVersionUID = 1L;

  public Personne() {
    super();
  }

  public int getId() {
    return this.id;
  }

  public void setId(int id) {
    this.id = id;
  }

  public String getPrenom() {
    return this.prenom;
  }

  public void setPrenom(String prenom) {
    this.prenom = prenom;
  }

  public String getNom() {
    return this.nom;
  }

  public void setNom(String nom) {
    this.nom = nom;
  }

}
```

JPA propose une API pour manipuler ces entités notamment en utilisant un objet de type EntityManager.

Exemple : rechercher et supprimer une occurrence

```
    private EntityManager em;
...
    Personne personne = em.find(Personne.class, 4);
    if (personne == null) {
      System.out.println("Personne non trouvée");
    } else {
      em.remove(personne);
    }
```

La mise en oeuvre de JPA est détaillée dans le chapitre «JPA (Java Persistence API)»

48.6.5. Des services web plus simple à écrire

La simplification de JAVA EE 5 concerne aussi les services web : les services web sont plus simples à développer et le nombre de standards supportés a augmenté. Cette simplification est largement assurée par l'utilisation des annotations et de comportement par défaut.

Exemple :

```
package com.jmdoudoux.test.jaxws;

import javax.jws.WebService;

@WebService
public class MonService {

  public String saluer(String param) {
    return "Bonjour " + param;
  }
}
```

Avec Java EE 5, l'utilisation d'annotations a grandement simplifié le développement de services web.

Java EE 5 propose aussi plusieurs API concernant les services web : Java API for XML-Based Web Services (JAX-WS) 2.0 (JSR 224), Java Architecture for XML Binding (JAXB) 2.0 (JSR 222) et Web Services Metadata for the Java Platform (JSR 181).

JAX-WS 2.0 est la nouvelle API pour le développement de services web. Elle succède à JAX-RPC 1.1. Cette nouvelle version propose :

- l'utilisation des annotations
- le binding des données grâce à JAXB 2.0
- le support de SOAP 1.1 et 1.2
- le support de MTOM/XOP pour l'encodage optimisé des attachments
- le support des services web de type REST

La définition d'un service web consiste à utiliser l'annotation @WebService sur une classe.

Par défaut, toutes les méthodes publiques sont exposées en tant qu'opérations du service web. L'utilisation des annotations permet de ne pas avoir à définir de fichier de déploiement.

Pour modifier le mapping par défaut, certaines annotations peuvent être utilisées.

Exemple :

```
package com.jmdoudoux.test.jaxws;

import javax.jws.WebService;
import javax.jws.WebMethod;

@WebService(name="MonServiceWS")
public class MonService {

  @WebMethod(operationName="direBonjour")
  public String saluer(String param) {
    return "Bonjour " + param;
  }
}
```

Par défaut, le WSDL d'un service web sera généré dynamiquement par le conteneur selon les spécifications de JAX-WS 2.

JAX-WS 2.0 propose aussi une API pour permettre l'appel de services web par un client de façon asynchrone. Cet appel asynchrone peut être utilisé avec n'importe quel service web : l'invocation de services web de façon asynchrone n'implique aucun traitement particulier côté serveur. C'est simplement l'invocation côté client qui est différente.

48.6.6. Le développement d'applications Web

Le framework Java Server Faces version 1.2 est inclus dans la plate-forme Java EE 5.

La bibliothèque JavaServer Pages Standard Tag Library (JSTL) est incluse dans la plate-forme Java EE 5. JSTL propose un langage d'expression pour faciliter la manipulation d'entités et un ensemble de tags personnalisés.

Les incompatibilités entre les langages d'expressions de la JSTL et des JSF ont été corrigées, ce qui leur permet d'être utilisés simultanément grâce à UEL (Unified EL).

48.6.7. Les autres fonctionnalités

La version de 2.0 de JAXB offre un support complet des schémas XML.

L'API Streaming API for XML (StAX) défini une méthode pour parser un document XML à partir d'événements. Son mode de fonctionnement est différent de SAX : avec SAX le développeur écrit du code pour répondre à des événements émis par le parser, avec StAX c'est le programme qui pilote le parser.

Remarque : ces deux API ont été ajoutées à la version 6 de Java SE.

48.6.8. L'installation du SDK Java EE 5 sous Windows

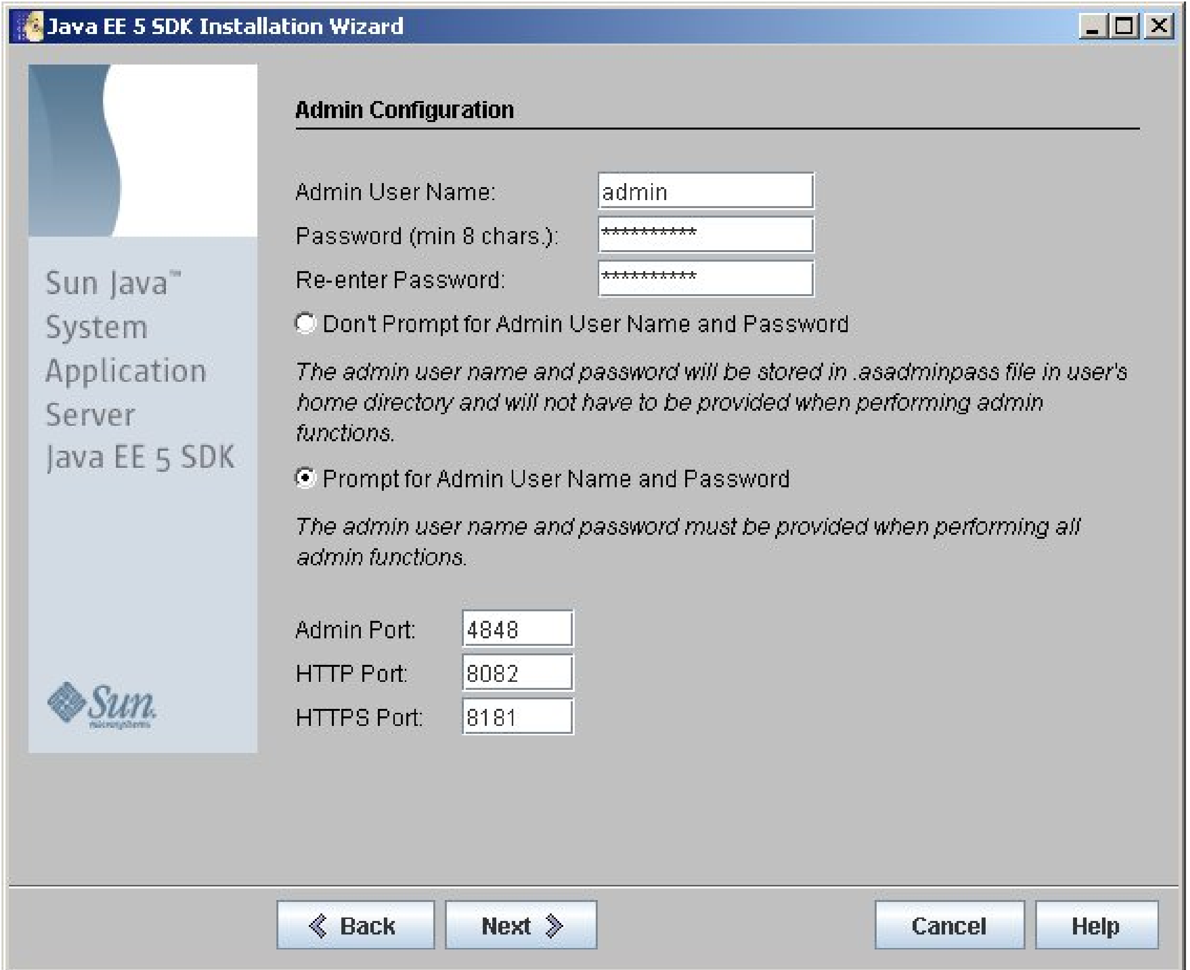
Téléchargez le fichier java_ee_sdk-5-windows.exe sur le site de Sun et exécutez le.

Les fichiers d'installation sont extraits

Extracting installation files...

Puis le programme d'installation est lancé pour guider l'utilisateur avec un assistant :

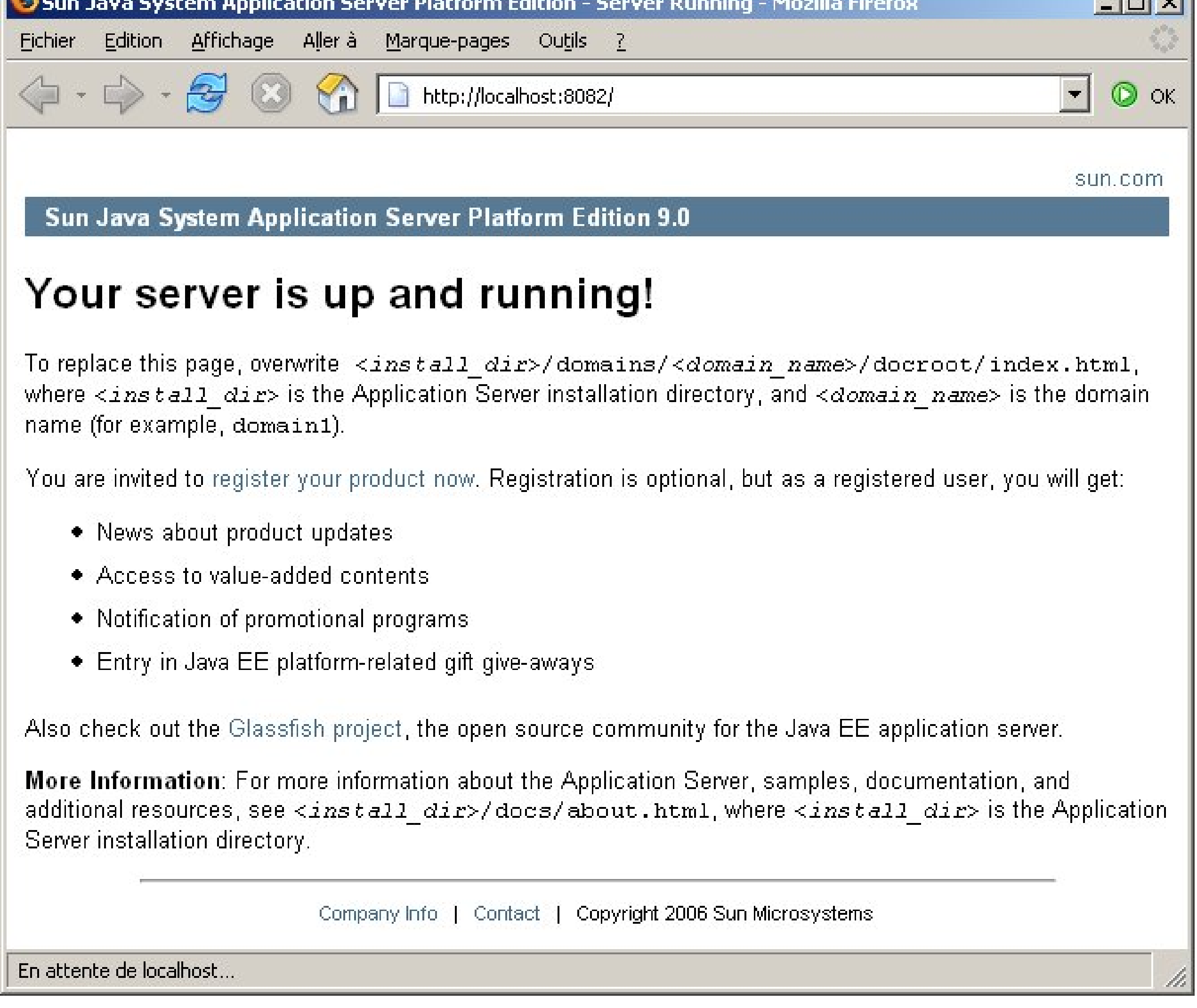
- Sur la page « Welcome », cliquez sur le bouton « Next »
- Sur la page « Software Licence Agreement », lisez la licence et si vous l'acceptez, cliquez sur « Yes » puis sur le bouton « Next »
- Sur la page « select Installation Directory, modifiez si nécessaire le répertoire d'installation puis cliquez sur le bouton « Next »
- Sur la page « Admin Configuration », saisissez le mot de passe, sélectionnez la saisie ou non du mot de passe, et modifiez les ports au besoin puis cliquez sur le bouton « Next »



- Sur la page « Installation options », cochez «Create desktop shortcut to autodeploy directory» et «create windows service» en plus des options déjà cochées, puis cliquez sur le bouton «Next». Le programme d'installation vérifie l'espace disque requis.
- Sur la page « Ready to Install », cliquez sur le bouton «Install Now».
- Le programme d'installation copie les fichiers.
- Sur la page « Installation Complete », cliquez sur le bouton « Start server »

Cliquez sur le bouton «OK» puis sur le bouton « Finish »

Ouvrez un navigateur sur l'url http://localhost:8082



48.7. La présentation de Java EE 6

La version 6 de la plate-forme Java EE est diffusée en décembre 2009. Elle s'appuie sur la plate-forme Java SE 6 dont elle utilise de nombreuses API (JDBC, JNDI, JAXB, JAXP, RMI, JMX, ...).

Cette nouvelle version de la plate-forme a plusieurs caractéristiques :

- Plus riche : de nouvelles spécifications sont ajoutées et les spécifications majeures sont enrichies
- Plus facile : généralisation de l'utilisation des POJO et des annotations notamment dans le tiers web
- Plus légère : création de la notion de profiles et de pruning, EJB Lite
- Toujours aussi robuste après 1O ans d'existence

L'implémentation de référence est le projet open source GlassFish version 3.

48.7.1. Les spécifications de Java EE 6

C'est la version de la plate-forme avec le plus grand nombres de spécifications. Elle contient 28 spécifications :

Technologies	JSR

Java Platform, Enterprise Edition 6 (Java EE 6) (avec Managed Beans 1.0)	JSR 316
Technologies relatives aux services web	
Java API for RESTful Web Services (JAX-RS) 1.1	JSR 311
Implementing Enterprise Web Services 1.3	JSR 109
Java API for XML-Based Web Services (JAX-WS) 2.2	JSR 224
Java Architecture for XML Binding (JAXB) 2.2	JSR 222
Web Services Metadata for the Java Platform	JSR 181
Java API for XML-Based RPC (JAX-RPC) 1.1	JSR 101
Java APIs for XML Messaging 1.3	JSR 67
Java API for XML Registries (JAXR) 1.0	JSR 93
Technologies relatives aux développements web	
Java Servlet 3.0	JSR 315
JavaServer Faces 2.0	JSR 314
JavaServer Pages 2.2/Expression Language 2.2	JSR 245
Standard Tag Library for JavaServer Pages (JSTL) 1.2	JSR 52
Debugging Support for Other Languages 1.0	JSR 45
Technologies relatives aux développements d'applications	
Contexts and Dependency Injection for Java (Web Beans 1.0)	JSR 299
Dependency Injection for Java 1.0	JSR 330
Bean Validation 1.0	JSR 303
EnterpriseJava Beans 3.1 (avec Interceptors 1.1)	JSR 318
Java EE Connector Architecture 1.6	JSR 322
Java Persistence 2.0	JSR 317
Common Annotations for the Java Platform 1.1	JSR 250
Java Message Service API 1.1	JSR 914
Java Transaction API (JTA) 1.1	JSR 907
JavaMail 1.4	JSR 919
Gestion et sécurité	
Java Authentication Service Provider Interface for Containers	JSR 196
Java Authorization Contract for Containers 1.3	JSR 115
Java EE Application Deployment 1.2	JSR 88
J2EE Management 1.1	JSR 77

48.7.2. Une plate-forme plus légère

Une critique récurrente de la plate-forme Java EE est qu'elle est très riche et donc complexe. La plupart des applications de petite taille ou de taille moyenne n'ont pas besoin de toutes les technologies proposées par la plate-forme.

Cette richesse est aussi lié au fait que certaines technologies sont obsolètes car remplacées par une nouvelle technologie : ces anciennes versions sont cependant conservées pour des raisons de compatibilité.

48.7.2.1. La notion de profile

Java EE 6 définit la notion de profile qui est un sous ensemble et/ou un sur ensemble de Java EE 6 pour des besoins particuliers.

Les profiles sont conçus pour proposer une solution technique particulière pour des besoins spécifiques. Cela permet à un fournisseur de n'avoir à proposer que le support des technologies incluses dans le profile plutôt que d'avoir à implémenter toutes celles de la plate-forme Java EE.

Java EE 6 définit un premier profile : le web profile qui se concentre sur le développement d'applications de type web. Il doit pouvoir être exécuté dans un simple conteneur web car le packaging ce fait dans une archive de type war.

Le web profile est composé de plusieurs spécifications pour définir un sous ensemble de Java EE :

Servlet 3.0	JSP 2.2	EL 1.2
JSTL 1.2	JSF 2.0	EJB Lite 3.1
JTA 1.1	JPA 2.0	Bean Validation 1.0
DI 1.0	CDI 1.0	Interceptors 1.1

D'autres profiles devraient être définis ultérieurement de façon indépendantes des évolutions de la plate-forme Java EE.

Un fournisseur n'a plus l'obligation de fournir une implémentation complète de la spécification Java EE mais peut simplement fournir une implémentation d'un profile. Bien sûre dans ce cas, les fonctionnalités utilisables sont uniquement définies dans ce profile.

Le notion de profile a fait l'objet de nombreux débats au sein des membres de la JSR notamment sur leur définition, sur leur compatibilité et sur la confusion et les interrogations qu'elle peut apporter chez les développeurs.

48.7.2.2. La notion de pruning

La plate-forme Java EE est devenue au fils des versions imposantes en terme de nombre de spécifications, de fonctionnalités et d'API. Aucunes des précédentes versions n'a supprimé de fonctionnalités même si certaines sont obsolètes car remplacées ou rarement adoptées ou utilisées. Cela pose plusieurs soucis :

- Pour le développeur : la taille du SDK augmente avec le nombre d'API
- Pour les fournisseurs de serveurs d'applications : l'obligation de support pour ces fonctionnalités avec un accroissement de la taille des serveurs, de leur consommation en ressources et de leur temps de démarrage

Certaines de ces fonctionnalités sont de plus obsolètes car remplacées une plus récente ou ne sont que peux ou pas utilisées.

Java EE 6 entame donc une cure d'amaigrissement de la plate-forme en définissant un ensemble d'API déclarées comme pruned.

Les fonctionnalités déclarées pruned dans Java EE 6 sont :

- Entity CMP 2.x : cette API est avantageusement remplacée par JPA
- JAX-RPC : cette API est avantageusement remplacée par JAX-WS
- JAX-R : cette API est très peux utilisée car l'utilisation de UDDI n'est pas trés répandue
- JSR 88 (Java EE Application Deployment) : cette API est très peux mise en oeuvre par les fournisseurs de serveurs d'applications

Elles doivent toujours être disponibles dans une implémentation de Java EE 6 mais elles seront certainement amenées à disparaitre dans la prochaine version de la plate-forme Java EE et leur support ne sera plus obligatoire dans les

implémentations des serveurs d'applications. Le concept de pruned est plus fort que le concept de deprecated de la plate-forme Java SE.

Il est donc raisonnable de ne plus commencer à utiliser ces fonctionnalités dans de nouveaux développements et de migrer progressivement les applications existantes qui les utilisent.

Le but est de simplifier le développement des prochaines versions des conteneurs des serveurs d'applications qui n'auront plus l'obligation de fournir une implémentation des fonctionnalités déclarées pruned.

48.7.3. Les évolutions dans les spécifications existantes

Plusieurs spécifications existantes dans la version 5 de Java EE 5 ont été mises à jour dans la plate-forme Java EE 6.

48.7.3.1. Servlet 3.0

Cette nouvelle version de l'API servlet a pour but de simplifier son utilisation. Comme pour la plupart des API de Java EE, cette simplification repose en grande partie sur l'utilisation d'annotations telles que @WebServlet, @ServletFilter, @WebServletContextListener, @InitParam, @WebFilter, ... pour déclarer des entités ce qui permet de rendre le descripteur de déploiement web.xml plus léger voir optionnel.

Bien qu'une servlet puisse être annotée avec @WebServlet, elle doit toujours hériter de la classe HttpServlet notamment pour permettre d'identifier de façon unique les méthodes à invoquer selon le type de requête http à traiter.

Même si la plupart des développements n'utilise plus directement cette API au profit de frameworks, les développeurs de ces derniers vont pouvoir utiliser les nouvelles fonctionnalités de la spécification.

Le fichier web.xml est rendu modulaire et peut être rédigé sous la forme de fragments qui seront agrégés et fusionnés par le conteneur au moment du déploiement.

Un fragment est une portion du descripteur de déploiement dont le tag racine est <web-fragment> qui peut contenir la définition de tout ou partie des entités configurables dans le descripteur de déploiement.

Ceci pourra ainsi permettre d'éviter d'avoir à déclarer des servlets ou des filtres d'un framework utilisé dans une webapp. Le framework pourra simplement définir le fragment pour que ces déclarations soient prises en compte par le conteneur.

Le conteneur recherche des fragments du fichier web.xml dans le classpath de la webapp (WEB-INF/classes et dans les fichiers jar du répertoire WEB-INF/lib) pour les agréger et composer le fichier web.xml.

Servlet 3.0 propose un support des invocations asynchrones en utilisant l'attribut asyncSupported=True de l'annotation @WebServlet. Une api dédiée est proposée pour gérer l'état d'une requête.

L'interface ServletContext propose des méthodes pour permettre d'enregistrer dynamiquement des servlets, des filtres et des listeners.

L'implémentation de référence est GlassFish v3.

48.7.3.2. JSF 2.0

Cette nouvelle version de JSF a pour but de simplifier son utilisation. Comme pour la plupart des API de Java EE, cette simplification repose en grande partie sur l'utilisation d'annotations telles que @ManagedBean, @ManagedProperty, @ApplicationScoped, @SessionScoped, @FacesValidator, @FacesConverter ... qui permettent de rendre le fichier faces-config.xml beaucoup plus petit.

JSF 2.0 utilise le projet open source Facelets comme technologie pour la partie vue : l'organisation du contenu des pages et des composants est facilité grâce à l'utilisation de Facelets. Le développement de composants a été grandement simplifié grâce à Facelets en utilisant la notion de composition qui met en oeuvre XHTML et un tag JSF.

JSF propose en standard un support de fonctionnalités mettant en oeuvre Ajax sur des fonctions Javascript standardisées contenues dans le fichier jsf.js que chaque implémentation doit fournir. Le cycle de vie de traitement d'une requête JSF a du être adapté pour les traitements Ajax en incorporant la notion de page partielle (partial page).

L'implémentation de référence est Mojorra.

48.7.3.3. Les EJB 3.1

La version 3.1 des EJB poursuit sur la simplicité de la version 3.0 tout en apportant de nouvelles fonctionnalités et un enrichissement des fonctionnalités existantes :

- Les interfaces Local sont optionnelles pour les EJB Session
- EJB Singleton : le conteneur garantie qu'une seule instance de l'EJB sera accessible par défaut de façon thread-safe. Il est cependant possible d'avoir un contrôle sur la gestion de la concurrence des accès.
- Les invocations asynchrones des session beans
- EJB Timer
- Packaging dans un war qui est utilisé notamment dans le web profile
- Les noms JNDI portables pour faciliter le déploiement sur plusieurs serveurs d'applications.
- Le conteneur embarqué pour exécuter des EJB sous la plate-forme Java SE
- EJB Lite qui est utilisé notamment dans le web profile en proposant d'une petite partie des fonctionnalités proposées par les EJB (Session Bean, transactions et sécurité) en occultant notamment les appels distants, les MDB et le scheduling.

L'implémentation de référence est GlassFish v3.

Le chapitre «Les EJB 3.1» contient une description détaillée de cette API.

48.7.3.4. JPA 2.0

JPA 2.0 a fait l'objet d'une spécification dédiée.

Les possibilités de mapping sont enrichies avec

- le support des collections qui ne représentent pas de relations avec des entités grâce à l'annotation @ElementCollection (mappe une collection d'éléments ou une collection de type Map dans une table dédiée)
- le support des relations unidirectionnelles one-to-many

La version 2.0 de JPA propose de nouvelles fonctionnalités manquantes dans la version précédente :

- Une gestion plus fine des verrous (locks) notamment avec le support des verrous pessimistes
- Une API pour coder les critères de recherche de façon dynamique sous la forme d'un graphe d'objets
- Une API simple pour la gestion du cache

L'implémentation de référence est EclipseLink.

48.7.3.5. JAX-WS 2.2

Cette API permet la mise en oeuvre de services web de type Soap. La version 2.2 est incluse dans Java EE 6.

L'implémentation de référence est Metro.

48.7.3.6. Interceptors 1.1

Ils peuvent être utilisés sur les EJB et les Managed Beans.

48.7.4. Les nouvelles API

De nouvelles API ont été ajoutées à la version 6 de la plate-forme Java EE.

48.7.4.1. JAX-RS 1.1

L'API JAX-RS permet de mettre en oeuvre des services web de type RestFul.

L'inclusion de JAX-RS 1.1 dans la plate-forme Java EE suit la tendance à l'adoption grandissante des services web de type REST.

JAX-RS utilise des annotations sur des POJO pour masquer la complexité de traitement des requêtes et de génération des réponses ce qui permet de simplifier leurs mises en oeuvre.

Plusieurs annotations sont définies :

- @Path permet de déterminer l'URL d'accès à la ressource
- @GET, @POST, @PUT and @DELETE permet de déterminer la méthode http utilisée pour accéder à la ressource
- @QueryParam, @PathParam, @CookieParam et @HeaderParam permettent d'extraire les valeurs de la requête http (paramètres, cookies, header)
- @Produces, @Consumes permet de préciser le format de restitution ou de consommation de la ressource

Elle repose sur l'utilisation de ces annotations sur un POJO.

La version utilisée dans Java EE 6 est la 1.1.

Exemple :

```
@Path("/helloworld")
public class HelloWorldRS {
  @GET
  @Produces("text/plain")
  public String saluer() {
  return "Hello World";
  }
}
```

Cette version permet une utilisation de l'API avec les EJB.

L'implémentation de référence est Jersey.

48.7.4.2. Contexte and Dependency Injection (WebBeans) 1.0

Le but de cette spécification est de faciliter les interactions entre la couche de présentation, la couche métier et la couche de persistance notamment à l'aide de beans qui pourront être utilisés par plusieurs couches.

Cette spécification a été influencée par plusieurs projets open source notamment JBoss Seam, Google Guice et Spring.

CDI a pour objectif de fournir une glue entre les couches mettant en oeuvre JSF, EJB et JPA. Elle permet notamment d'enregistrer et de gérer des EJB, des entités JPA et des ManagedBeans sous la forme de composants qui seront injectables et utilisables grâce à EL.

CDI peut remplacer les backing beans de JSF.

La gestion du cycle de vie des composants par CDI se fait par rapport à un contexte (requête, session et application mais aussi deux nouveaux contextes nommé dependent et conversation).

L'implémentation de référence est JBoss Seams.

48.7.4.3. Dependency Injection 1.0

Le but de cette JSR est de proposer la standardisation d'un ensemble d'annotations utilisables avec n'importe quel moteur d'injection : le but n'est pas de spécifier un tel moteur.

Elle définit plusieurs annotations :

- @Inject : identifier un constructeur, une méthode ou un champ injectable
- @Named : permet de qualifier une dépendance avec une chaine de caractères
- @Qualifier : permet de qualifier une dépendance
- @Scope : permet de définir la portée de l'injection
- @Singleton : injection d'une instance unique

Google Guice et Spring 3.0 implémentent cette spécification.

48.7.4.4. Bean Validation 1.0

Cette API standardise la validation de données.

Les contraintes sont définies dans les beans avec des annotations (@NotNull, @Size, @Past, ...)

Bean Validation propose une API pour valider des données, définir ces propres contraintes et rechercher des contraintes.

Cette API est utilisée notamment par JSF 2.0 et JPA 2.0

L'implémentation de référence est Hibernate Validator 4.0.

Le chapitre «La validation des données» contient une description détaillée de cette API.

48.7.4.5. Managed Beans 1.0

C'est un modèle de composants légers : ce sont des POJO gérés par le conteneur.

Les managed beans supportent plusieurs services :

- Injection de dépendances : @Resource, @Inject
- Gestion du cycle de vie : @PostConstruct, @PreDestroy
- Intercepteurs : @Interceptor, @AroundInvoke

Un managed bean est un POJO annoté avec l'annotation @javax.annotation.ManagedBean. Cette annotation est issue de la JSR 250 (Commons annotations).

49. JavaMail

Le courrier électronique repose sur le concept du client/serveur. Ainsi, l'utilisation d'e mail requiert deux composants :

- un client de mail (Mail User Agent : MUA) tel que Outlook, Messenger, Eudora, ...
- un serveur de mail (Mail Transport Agent : MTA) tel que SendMail

Les clients de mail s'appuient sur un serveur de mail pour obtenir et envoyer des messages. Les échanges entre client et serveur sont normalisés par des protocoles particuliers.

JavaMail est une API qui permet d'utiliser le courrier électronique (e-mail) dans une application écrite en java (application cliente, applet, servlet, EJB, ...). Son but est d'être facile à utiliser, de fournir une souplesse qui permette de la faire évoluer et de rester le plus indépendant possible des protocoles utilisés.

JavaMail est une extension au JDK qui n'est donc pas fournie avec J2SE. Pour l'utiliser, il est possible de la télécharger sur le site de SUN : http://java.sun.com/products/javamail. Elle est intégrée au J2EE.

Les classes et interfaces sont regroupées dans quatre packages : javax.mail, javax.mail.event, javax.mail.internet, javax.mail.search.

Il existe deux versions de cette API :

- 1.1.3 : version fournie avec J2EE 1.2
- 1.2 : version courante

Les deux versions fonctionnent avec un JDK dont la version est au moins 1.1.6.

Cette API permet une abstraction assez forte de tout système de mail, ce qui lui permet d'ajouter des protocoles non gérés en standard. Pour gérer ces différents protocoles, il faut utiliser une implémentation particulière pour chacun d'eux, fournis par des fournisseurs tiers. En standard, JavaMail 1.2 fournit une implémentation pour les protocoles SMTP, POP3 et IMAP4. JavaMail 1.1.3 ne fournit une implémentation que pour les protocoles SMTP et IMAP : l'implémentation pour le protocole POP3 doit être téléchargée séparément.

Ce chapitre contient plusieurs sections :

- ♦ Le téléchargement et l'installation
- ♦ Les principaux protocoles
- ♦ Les principales classes et interfaces de l'API JavaMail
- ♦ L'envoi d'un e mail par SMTP
- ♦ La récupération des messages d'un serveur POP3
- ♦ Les fichiers de configuration

49.1. Le téléchargement et l'installation

Pour le J2SE, il est nécessaire de télécharger les fichiers utiles et de les installer.

Pour les deux versions de l'API, il faut télécharger la version correspondante, décompresser le fichier dans un répertoire

et ajouter le fichier mail.jar dans le CLASSPATH.

Ensuite il faut aussi installer le framework JAF (Java Activation Framework) : télécharger le fichier, décompresser et ajouter le fichier activation.jar dans le CLASSPATH

Pour pouvoir utiliser le protocole POP3 avec JavaMail 1.1.3, il faut télécharger en plus l'implémentation de ce protocole et inclure le fichier POP3.jar dans le CLASSPATH.

Pour le J2EE 1.2.1, l'API version 1.1.3 est intégrée à la plate-forme. Elle ne contient donc pas l'implémentation pour le protocole POP3. Il faut la télécharger et l'installer en plus comme avec le J2SE.

Pour le J2EE 1.3, il n'y a rien de particulier à faire puisque l'API version 1.2 est intégrée à la plate-forme.

49.2. Les principaux protocoles

49.2.1. Le protocole SMTP

SMTP est l'acronyme de Simple Mail Transport Protocol. Ce protocole défini par la recommandation RFC 821 permet l'envoi de mails vers un serveur de mails qui supporte ce protocole.

49.2.2. Le protocole POP

POP est l'acronyme de Post Office Protocol. Ce protocole défini par la recommandation RFC 1939 permet la réception de mails à partir d'un serveur de mails qui supporte ce protocole. La version courante de ce protocole est 3. C'est un protocole très populaire sur Internet. Il définit une boite aux lettres unique pour chaque utilisateur. Une fois que le message est reçu par le client, il est effacé du serveur.

49.2.3. Le protocole IMAP

IMAP est l'acronyme de Internet Message Acces Procol. Ce protocole défini par la recommandation RFC 2060 permet aussi la réception de mail à partir d'un serveur de mail qui supporte ce protocole. La version courante de ce protocole est 4. Ce protocole est plus complexe car il apporte des fonctionnalités supplémentaires : plusieurs répertoires par utilisateur, partage de répertoire entre plusieurs utilisateurs, maintient des messages sur le serveur, etc ...

49.2.4. Le protocole NNTP

NNTP est l'acronyme de Network News Transport Protocol. Ce protocole est utilisé par les forums de discussion (news).

49.3. Les principales classes et interfaces de l'API JavaMail

JavaMail propose des classes et interfaces qui encapsulent ou définissent les objets liés à l'utilisation des mails et les protocoles utilisés pour les échanger.

49.3.1. La classe Session

La classe Session encapsule pour un client donné sa connexion avec le serveur de mail. Cette classe encapsule les données liées à la connexion (options de configuration et données d'authentification). C'est à partir de cet objet que toutes les actions concernant les mails sont réalisées.

Les paramètres nécessaires sont fournis dans un objet de type Properties. Un objet de ce type est utilisé pour contenir les variables d'environnements : placer certaines informations dans cet objet permet de partager des données.

Une session peut être unique ou partagée par plusieurs entités.

Exemple :

```
// creation d'une session unique
Session session = Session.getInstance(props,authenticator);
// creation d'une session partagee
Session defaultSession = Session.getDefaultInstance(props,authenticator);
```

Pour obtenir une session, deux paramètres sont attendus :

- un objet Properties qui contient les paramètres d'initialisation. Un tel objet est obligatoire
- un objet Authenticator optionnel qui permet d'authentifier l'utilisateur auprès du serveur de mail

La méthode setDebug() qui attend en paramètre un booléen est très pratique pour debugger car avec le paramètre true, elle affiche des informations lors de l'utilisation de la session notamment le détail des commandes envoyées au serveur de mail.

49.3.2. Les classes Address, InternetAddress et NewsAddress

La classe Address est une classe abstraite dont héritent toutes les classes qui encapsulent une adresse dans un message.

Deux classes filles sont actuellement définies :

- InternetAddress
- NewsAddress

Le classe InternetAddress encapsule une adresse email respectant le format de la RFC 822. Elle contient deux champs : address qui contient l'adresse e mail et personal qui contient le nom de la personne. La classe possède des constructeurs, des getters et des setters pour utiliser ces attributs.

Le plus simple pour créer un objet InternetAddress est d'appeler le constructeur en lui passant en paramètre une chaîne de caractère contenant l'adresse e-mail.

Exemple :

```
InternetAddress vInternetAddresses = new InternetAddress;
vInternetAddresses = new InternetAddress("moi@chez-moi.fr");
```

Un second constructeur permet de préciser l'adresse e-mail et un nom en clair.

La méthode getLocalAddress(Session) permet de déterminer si possible l'objet InternetAddress encapsulant l'adresse e mail de l'utilisateur courant, sinon elle renvoie null.

La méthode parse(String) permet de créer un tableau d'objet InternetAddress à partir d'une chaîne contenant les adresses e mail séparées par des virgules.

Un objet InternetAddress est nécessaire pour chaque émetteur et destinataire du mail. L'API ne vérifie pas l'existence des adresses fournies. C'est le serveur de mail qui vérifiera les destinataires et éventuellement les émetteurs selon son

paramétrage.

La classe NewsAddress encapsule une adresse news (forum de discussion) respectant le format RFC1036. Elle contient deux champs : host qui contient le nom du serveur et newsgroup qui le nom du forum

La classe possède des constructeurs, des getters et des setters pour utiliser ces attributs.

49.3.3. L'interface Part

Cette interface définit un certain nombre d'attributs commun à la plupart des systèmes de mail et un contenu.

Le contenu peut être renvoyé sous trois formes : DataHandler, InputStream et Object.

Cette interface définit plusieurs méthodes principalement des getters et des setters dont les principaux sont :

Méthode	Rôle
int getSize()	Renvoie la taille du contenu sinon -1 si elle ne peut être déterminée
int getLineCount()	Renvoie le nombre de ligne du contenu sinon -1 s'il ne peut être déterminé
String getContentType()	Renvoie le type du contenu sinon null
String getDescription()	Renvoie la description
void setDescription(String)	Mettre à jour la description
InputStream getInputStream()	Renvoie le contenu sous la forme d'un flux
DataHandler getDataHandler()	Renvoie le contenu sous la forme d'un objet DataHandler
Object getContent()	Renvoie le contenu sous la forme d'un objet. Un cast est nécessaire selon le type du contenu.
void setText(String)	Mettre à jour le contenu sous forme d'une chaîne de caractères fournie en paramètre

49.3.4. La classe Message

La classe abstraite Message encapsule un Message. Le message est composé de deux parties :

- une en-tête qui contient des attributs
- un corps qui contient les données à envoyer

Pour la plupart de ces données, la classe Message implémente l'interface Part qui encapsule les attributs nécessaires à la distribution du message (auteur, destinataire, sujet ...) et le corps du message.

Le contenu du message est stocké sous forme d'octet. Pour accéder à son contenu, il faut utiliser un objet du JavaBean Activation Framework (JAF) : DataHandler. Ceci permet une séparation des données nécessaires à la transmission et du contenu du message qui peut ainsi prendre n'importe quel format. La classe Message ne connaît pas directement le type du contenu du corps du message.

JavaMail fourni en standard une classe fille nommée MimeMessage qui implémente la recommandation RFC 822 pour les messages possédant un type Mime.

Il y a deux façons d'obtenir un objet de type Message : instancier une classe fille pour créer un nouveau message ou utiliser un objet de type Folder pour obtenir un message existant.

La classe Message défini deux constructeurs en plus du constructeur par défaut :

Constructeur	Rôle
Message(session)	Créer un nouveau message
Message(Folder, int)	Créer un message à partir d'un message existant

La classe MimeMessage est la seule classe fille qui hérite de la classe Message. Elle dispose de plusieurs constructeurs.

Exemple :

```
MimeMessage message = new MimeMessage(session);
```

Elle possède de nombreuses méthodes pour initialiser les données du message :

Méthode	Rôle
void addFrom(Address[])	Ajouter des émetteurs au message
void addRecipient(RecepientType, Address[])	Ajouter des destinataires à un type (direct, en copie ou en copie cachée)
Flags getFlags()	Renvoie les états du message
Adress[] getFrom()	Renvoie les émetteurs
int getLineCount()	Renvoie le nombre ligne du message
Address[] getRecipients(RecepientType)	Renvoie les destinataires du type fourni en paramètre
Address getReplyTo()	Renvoie les emails pour la réponse
int getSize()	Renvoie la taille du message
String getSubject()	Renvoie le sujet
Message reply(boolean)	Créer un message pour la réponse : le booléen indique si la réponse ne doit être faite qu'à l'émetteur
void set content(Object, String)	Mettre à jour le contenu du message en précisant son type mime
void setFrom(Address)	Mettre à jour l'émetteur
void setRecipients(RecepientsType, Address[])	Mettre à jour les destinataires d'un type
void setSendDate(Date)	Mettre à jour la date d'envoi
void setText(String)	Mettre à jour le contenu du message avec le type mime « text/plain »
void setReply(Address)	Mettre à jour le destinataire de la réponse
void writeTo(OutputStream)	Envoie le message au format RFC 822 dans un flux. Très pratique pour visualiser le message sur la console en passant en paramètre (System.out)

La méthode addRecipient() permet d'ajouter un destinataire et le type d'envoi.

Le type d'envoi est précisé grâce une constante pour chaque type :

- destinataire direct : Message.RecipientType.TO
- copie conforme : Message.RecipientType.CC
- copie cachée : Message.RecipientType.BCC

La méthode setText() permet de facilement mettre une chaîne de caractères dans le corps du message avec un type MIME « text/plain ». Pour envoyer un message dans un format différent, par exemple HTML, il utilise la méthode

setContent() qui attend en paramètre un objet et une chaîne qui contient le type MIME du message.

Exemple :

```
String texte = "<H1>bonjour</H1><a
    href=\"mailto:moi@moi.fr\">mail</a>";
message.setContent(texte, "text/html");
```

Il est possible de joindre avec le mail des ressources sous forme de pièces jointes (attachments). Pour cela, il faut :

- instancier un objet de type MimeMessage
- renseigner les éléments qui composent l'en-tête : émetteur, destinataire, sujet ...
- Instancier un objet de type MimeMultiPart
- Instancier un objet de type MimeBodyPart et alimenter le contenu de l'élément
- Ajouter cet objet à l'objet MimeMultiPart grâce à la méthode addBodyPart()
- Répéter l'instanciation et l'alimentation pour chaque ressource à ajouter
- utiliser la méthode setContent() du message en passant en paramètre l'objet MimeMultiPart pour associer le message et les pièces jointes au mail

Exemple :

```
Multipart multipart = new MimeMultipart();

// creation partie principale du message
BodyPart messageBodyPart = new MimeBodyPart();
messageBodyPart.setText("Test");
multipart.addBodyPart(messageBodyPart);

// creation et ajout de la piece jointe
messageBodyPart = new MimeBodyPart();
DataSource source = new FileDataSource("image.gif");
messageBodyPart.setDataHandler(new DataHandler(source));
messageBodyPart.setFileName("image.gif");
multipart.addBodyPart(messageBodyPart);

// ajout des éléments au mail
message.setContent(multipart);
```

49.3.5. Les classes Flags et Flag

Cette classe encapsule un ensemble d'états pour un message.

Il existe deux types d'états : les états prédéfinis (System Flag) et les états particuliers définis par l'utilisateur (User Defined Flag)

Un état prédéfini est encapsulé par la classe Internet Flags.Flag. Cette classe définit plusieurs états statiques :

Etat	Rôle
Flags.Flag.ANSWERED	Le message a été demandé : positionné par le client
Flags.Flag.DELETED	Le message est marqué pour la suppression
Flags.Flag.DRAFT	Le message est un brouillon
Flags.Flag.FLAGGED	Le message est marqué dans un état qui n'a pas de définition particulière
Flags.Flag.RECENT	Le message est arrivé récemment. Le client ne peut pas modifier cet état.
Flags.Flag.SEEN	Le message a été visualisé : positionné à l'ouverture du message
Flags.Flag.USER	Le client a la possibilité d'ajouter des états particuliers

Tous ces états ne sont pas obligatoirement supportés par le serveur.

La classe Message possède plusieurs méthodes pour gérer les états d'un message. La méthode getFlags() renvoie un objet Flag qui contient les états du message. Les méthodes setFlag(Flag, boolean) permettent d'ajouter un état du message. La méthode contains(Flag) vérifie si l'état fourni en paramètre est positionné pour le message.

La classe Flags possède plusieurs méthodes pour gérer les états dont les principales sont :

Méthode	Rôle
void add(Flags.Flag)	Permet d'ajouter un état
void add(Flags)	Permet d'ajouter un ensemble d'état
void remove(Flags.Flag)	Permet d'enlever un état
void remove(Flags)	Permet d'enlever un ensemble d'état
boolean contains(Flags.Flag)	Permet de savoir si un état est positionné

49.3.6. La classe Transport

La classe Transport se charge de réaliser l'envoi du message avec le protocole adéquat. C'est une classe abstraite qui contient la méthode static send() pour envoyer un mail.

Il est possible d'obtenir un objet Transport dédié au protocole particulier utilisé par la session en utilisant la méthode getTransport() d'un objet Session. Dans ce cas, il faut :

1. établir la connexion en utilisant la méthode connect() avec le nom du serveur, le nom de l'utilisateur et son mot de passe
2. envoyer le message en utilisant la méthode sendMessage() avec le message et les destinataires. La méthode getAllRecipients() de la classe message permet d'obtenir ceux contenus dans le message.
3. fermer la connexion en utilisant la méthode close()

Il est préférable d'utiliser une instance de Transport tel qu'expliqué ci dessus lorsqu'il y a plusieurs mails à envoyer car on peut maintenir la connexion avec le serveur ouverte pendant les envois.

La méthode static send() ouvre et ferme la connexion à chacun de ces appels.

49.3.7. La classe Store

La classe abstraite store qui représente un système de stockage de messages. Pour obtenir une instance de cette classe, il faut utiliser la méthode getStore() d'un objet de type Session en lui donnant comme paramètre le protocole utilisé.

Pour pouvoir dialoguer avec le serveur de mail, il faut appeler la méthode connect() en lui précisant le nom du serveur, le nom d'utilisateur et le mot de passe de l'utilisateur.

La méthode close() permet de libérer la connexion avec le serveur.

49.3.8. La classe Folder

La classe abstraite Folder représente un répertoire dans lequel les messages sont stockés. Pour obtenir une instance de cette classe, il faut utiliser la méthode getFolder() d'un objet de type Store en lui précisant le nom du répertoire.

Avec le protocole POP3 qui ne gère qu'un seul répertoire, le seul possible est « INBOX ».

Pour pouvoir être utilisé, il faut appeler la méthode open() de la classe Folder en lui précisant le mode d'utilisation : READ_ONLY ou READ_WRITE.

Pour obtenir les messages contenus dans le répertoire, il faut appeler la méthode getMessages(). Cette méthode renvoie un tableau de Message qui peut être null si aucun message n'est renvoyé.

Une fois les opérations terminées, il faut fermer le répertoire en utilisant la méthode close().

49.3.9. Les propriétés d'environnement

JavaMail utilise des propriétés d'environnement pour recevoir certains paramètres de configuration. Ils sont stockés dans un objet de type Properties.

L'objet Properties peut contenir un certains nombre de propriétés qui possèdent des valeurs par défaut :

Propriété	Rôle	Valeur par défaut
mail.store.protocol	Protocole de stockage du message	le premier protocole concerné dans le fichier de configuration
mail.transport.protocol	Protocole de transport par défaut	le premier protocole concerné dans le fichier de configuration
mail.host	Serveur de Mail par défaut	localhost
mail.user	Nom de l'utilisateur pour se connecter au serveur de mail	user.name
mail.protocol.host	Serveur de mail pour un protocole dédié	mail.host
mail.protocol.user	Nom de l'utilisateur pour se connecter au serveur de mail pour un protocole dédié	
mail.from	adresse par défaut de l'expéditeur	user.name@host
mail.debug	mode de débogage par défaut	

Attention : l'utilisation de JavaMail dans une applet implique de fournir explicitement toutes les valeurs des propriétés utiles car une applet n'a pas la possibilité de définir toutes les valeurs par défaut car l'accès à ces propriétés est restreint.

L'usage de certains serveurs de mail nécessite l'utilisation d'autres propriétés.

49.3.10. La classe Authenticator

Authenticator est une classe abstraite qui propose des méthodes de base pour permettre d'authentifier un utilisateur. Pour l'utiliser, il faut créer une classe fille qui se chargera de collecter les informations. Plusieurs méthodes appelées selon les besoins sont à redéfinir :

Méthode	Rôle
String getDefaultUserName()	
PasswordAuthentication getPasswordAuthentication()	
int getRequestingPort()	
String getRequestingPort()	

String getRequestingProtocol()	
InetAddress getRequestingSite()	

Par défaut, la méthode getPasswordAuthentication() de la classe Authentication renvoie null. Cette méthode renvoie un objet PasswordAuthentication à partir d'une source de données (boîte de dialogue pour saisie, base de données, ...).

Une instance d'une classe fille de la classe Authenticator peut être fournie à la session. L'appel à Authenticator sera fait selon les besoins par la session.

49.4. L'envoi d'un e mail par SMTP

Pour envoyer un e mail via SMTP, il faut suivre les principales étapes suivantes :

- Positionner les variables d'environnement nécessaires
- Instancier un objet Session
- Instancier un objet Message
- Mettre à jour les attributs utiles du message
- Appeler la méthode send() de la classe Transport

Exemple :

```
import javax.mail.internet.*;
import javax.mail.*;
import java.util.*;

/**
 * Classe permettant d'envoyer un mail.
 */
public class TestMail {
   private final static String MAILER_VERSION = "Java";
   public static boolean envoyerMailSMTP(String serveur, boolean debug) {
         boolean result = false;
         try {
                  Properties prop = System.getProperties();
                  prop.put("mail.smtp.host", serveur);
                  Session session = Session.getDefaultInstance(prop,null);
                  Message message = new MimeMessage(session);
                  message.setFrom(new InternetAddress("moi@chez-moi.fr"));
                  InternetAddress[] internetAddresses = new InternetAddress[1];
                  internetAddresses[0] = new InternetAddress("moi@chez-moifr");
                  message.setRecipients(Message.RecipientType.TO,internetAddresses);
                  message.setSubject("Test");
                  message.setText("test mail");
                  message.setHeader("X-Mailer", MAILER_VERSION);
                  message.setSentDate(new Date());
                  session.setDebug(debug);
                  Transport.send(message);
                  result = true;
         } catch (AddressException e) {
                  e.printStackTrace();
         } catch (MessagingException e) {
                  e.printStackTrace();
         }
         return result;
   }

   public static void main(String[] args) {
         TestMail.envoyerMailSMTP("10.10.50.8",true);
   }
}
```

javac -classpath activation.jar;mail.jar;smtp.jar %1.java

java -classpath .;activation.jar;mail.jar;smtp.jar %1

49.5. La récupération des messages d'un serveur POP3

Cette section sera développée dans une version future de ce document

49.6. Les fichiers de configuration

Ces fichiers permettent d'enregistrer des implémentations de protocoles supplémentaires et des valeurs par défaut. Il existe 4 fichiers répartis en deux catégories :

- javamail.providers et javamail.default.providers
- javamail.address.map et javamail.default.address.map

JavaMail recherche les informations contenues dans ces fichiers dans l'ordre suivant :

1. $JAVA_HOME/lib

2. META-INF/javamail.xxx dans le fichier jar de l'application

3. MATA-INF/javamail.default.xxx dans le fichier jar de javamail

Il est ainsi possible d'utiliser son propre fichier sans faire de modification dans le fichier jar de JavaMail. Cette utilisation peut se faire sur le poste client ou dans le fichier jar de l'application, ce qui offre une grande souplesse.

49.6.1. Les fichiers javamail.providers et javamail.default.providers

Ce sont deux fichiers au format texte qui contiennent la liste et la configuration des protocoles dont le système dispose d'une implémentation. L'application peut ainsi rechercher la liste des protocoles utilisables.

Chaque protocole est défini en utilisant des attributs avec la forme nom=valeur suivi d'un point virgule. Cinq attributs sont définis (leur nom doit être en minuscule) :

Nom de l'attribut	Rôle	Présence
protocol	nom du protocole	obligatoire
type	type protocole : « store » ou « transport »	obligatoire
class	nom de la classe contenant l'implémentation du protocole	obligatoire
vendor	nom du fournisseur	optionnelle
version	numéro de version	optionnelle

Exemple : le contenu du fichier META-INF/javamail.default.providers

```
# JavaMail IMAP provider Sun Microsystems, Inc
```

```
protocol=imap; type=store; class=com.sun.mail.imap.IMAPStore; vendor=Sun Microsystems, Inc;
# JavaMail SMTP provider Sun Microsystems, Inc
protocol=smtp; type=transport; class=com.sun.mail.smtp.SMTPTransport; vendor=Sun Microsystems,
 Inc;
# JavaMail POP3 provider Sun Microsystems, Inc
protocol=pop3; type=store; class=com.sun.mail.pop3.POP3Store; vendor=Sun Microsystems, Inc;
```

49.6.2. Les fichiers javamail.address.map et javamail.default.address.map

Ce sont deux fichiers au format texte qui permettent d'associer un type de transport avec un protocole. Cette association se fait sous la forme nom=valeur suivi d'un point virgule.

Exemple : le contenu du fichier META-INF/javamail.default.address.map

```
rfc822=smtp
```

50. JMS (Java Messaging Service)

JMS, acronyme de Java Messaging Service, est une API fournie par Sun pour permettre un dialogue standard entre des applications ou des composants via des brokers de messages ou MOM (Middleware Oriented Messages). Elle permet donc d'utiliser des services de messaging dans des applications Java comme le fait l'API JDBC pour les bases de données.

La page officielle de JMS est à l'URL : http://java.sun.com/products/jms/index.htm

Ce chapitre contient plusieurs sections :

- La présentation de JMS
- Les services de messages
- Le package javax.jms
- L'utilisation du mode point à point (queue)
- L'utilisation du mode publication/abonnement (publish/subscribe)
- La gestion des erreurs
- JMS 1.1
- Les ressources relatives à JMS

50.1. La présentation de JMS

JMS a été intégré à la plateforme J2EE à partir de la version 1.3 mais il n'existe pas d'implémentation officielle de cette API avant la version 1.3 du J2EE. JMS est utilisable avec les versions antérieures mais elle oblige à utiliser un outil externe qui implémente l'API.

Chaque fournisseur (provider) doit fournir une implémentation de ces spécifications. Il existe un certain nombre d'outils qui implémentent JMS dont la majorité sont des produits commerciaux.

Dans la version 1.3 du J2EE, JMS peut être utilisé dans un composant web ou un EJB, un type d'EJB particulier a été ajouté pour traiter les messages et des échanges JMS peuvent être intégrés dans une transaction gérée avec JTA (Java Transaction API).

JMS définit plusieurs entités :

- Un provider JMS : outil qui implémente l'API JMS pour échanger les messages : ce sont les brokers de messages
- Un client JMS : composant écrit en java qui utilise JMS pour émettre et/ou recevoir des messages.
- Un message : données échangées entre les composants

Différents objets utilisés via JMS sont généralement stockés dans l'annuaire JNDI du serveur d'application ou du provider du MOM :

- La fabrique de connexion (ConnectionFactory)
- Les destinations à utiliser (Queue et Topic)

JMS définit deux modes pour la diffusion des messages

* Point à point (Point to point) : dans ce mode un message est envoyé par un producteur et est reçu par un unique consommateur. Le support utilisé pour la mise en oeuvre de ce mode est la file (queue). Le message émis est stocké dans la file jusqu'à ce que le consommateur le lise dans la file et envoi une notification de réception du message. A ce moment là le message est supprimé de la file. Le message a généralement une date d'expiration.
* Publication / souscription (publish/subscribe) : dans ce mode un message est envoyé par un producteur et est reçu par un ou plusieurs consommateurs. Le support utilisé pour la mise en oeuvre de ce mode est le sujet (topic). Chaque consommateur doit s'abonner un à sujet (souscription). Seuls les messages émis à partir de cet abonnement sont accessibles par le consommateur.

Dans la version 1.0 de JMS, ces modes utilisent des interfaces distinctes.
Dans la version 1.1 de JMS, ces interfaces sont toujours utilisables mais il est aussi possible d'utiliser des interfaces communes à ces modes ce qui les rend interchangeables.

Les messages sont asynchrones mais JMS définit deux modes pour consommer un message :

* Mode synchrone : ce mode nécessite l'appel de la méthode receive() ou d'une de ces surcharges. Dans ce cas, l'application est arrêtée jusqu'à l'arrivée du message. Une version surchargée de cette méthode permet de rendre la main après un certain timeout.
* Mode asynchrone : il faut définir un listener qui va lancer un thread qui va attendre les messages et exécuter une méthode lors de leur arrivée.

JMS propose un support pour différents types de messages : texte brut, flux d'octets, objets java sérialisés, ...

50.2. Les services de messages

Les brokers de messages ou MOM (Middleware Oriented Message) permettent d'assurer l'échange de messages entre deux composants nommés clients. Ces échanges peuvent se faire dans un contexte interne (pour l'EAI) ou un contexte externe (pour le B2B).

Les deux clients n'échangent pas directement des messages : un client envoi un message et le client destinataire doit demander la réception du message. Le transfert du message et sa persistance sont assurés par le broker.

Les échanges de message sont :

* asynchrones :
* fiables : les messages ne sont délivrés qu'un et une seule fois

Les MOM représentent le seul moyen d'effectuer un échange de messages asynchrones. Ils peuvent aussi être très pratiques pour l'échange synchrone de messages plutôt que d'utiliser d'autres mécanismes plus compliqués à mettre en oeuvre (sockets, RMI, CORBA ...).

Les brokers de messages peuvent fonctionner selon deux modes :

* le mode point à point (point to point)
* le mode publication/abonnement (publish/subscribe)

Le mode point à point (point to point) repose sur le concept de files d'attente (queues). Le message est stocké dans une file d'attente puis il est lu dans cette file ou dans une autre. Le transfert du message d'une file à l'autre est réalisé par le broker de message.

Chaque message est envoyé dans une seule file d'attente. Il y reste jusqu'à ce qu'il soit consommé par un client et un seul. Le client peut le consommer ultérieurement : la persistance est assurée par le broker de message.

Le mode publication/abonnement repose sur le concept de sujets (Topics). Plusieurs clients peuvent envoyer des messages dans ce topic. Le broker de message assure l'acheminement de ce message à chaque client qui se sera préalablement abonné à ce topic. Le message possède donc potentiellement plusieurs destinataires. L'émetteur du message ne connaît pas les destinataires qui se sont abonnés.

Les principaux brokers de messages commerciaux sont :

Produit	Société	URL
Sonic MQ	Progress software	http://www.progress.com/sonicmq/index.htm
VisiMessage	Borland	http://www.borland.com/appserver
Swift MQ		http://www.swiftmq.com
MessageQ	BEA	http://www.bea.com/products/messageq/datasheet.shtml
Websphere MQ (MQ Series)	IBM	http://www-4.ibm.com/software/ts/mqseries
Rendez vous	Tibco	http://www.tibco.com/products/rv/index.html

Il existe quelques brokers de messages open source :

Outils	URL
OpenJMS	http://openjms.sf.net/
Joram	http://joram.objectweb.org/

50.3. Le package javax.jms

Ce package et ses sous packages contiennent plusieurs interfaces qui définissent l'API.

- Connection
- Session
- Message
- MessageProducer
- MessageListener

50.3.1. La fabrique de connexion

Un objet de type factory est un objet qui permet de retourner un objet pour se connecter au broker de messages. Il faut fournir plusieurs paramètres à l'objet de type factory.

Il existe deux types de factory : QueueConnectionFactory et TopicConnectionFactory selon le type d'échanges que l'on fait. Ce sont des interfaces que le broker de message doit implémenter pour fournir des objets.

Pour obtenir un objet de ce type, il faut soit instancier directement un tel objet soit faire appel à JNDI pour l'obtenir. Cette dernière solution est préférable car elle est plus portable.

La fabrique de type ConnectionFactory permet d'obtenir une instance de l'interface Connection. Cette instance est du type de l'implémentation fournie par le provider, ce qui permet de proposer une manière unique d'obtenir une instance de chaque implémentation.

Chaque provider fourni sa propre solution pour gérer ces objets contenus dans l'annuaire JNDI.

50.3.2. L'interface Connection

Cette interface définit des méthodes pour la connexion au broker de messages.

Cette connexion doit être établie en fonction du mode utilisé :

* l'interface QueueConnection pour le mode point à point
* l'interface TopicConnection pour le mode publication/abonnement

Pour obtenir l'un ou l'autre, il faut utiliser un objet de type factory correspondant au type QueueConnectionFactory ou TopicConnectionFactory avec la méthode correspondante : createQueueConnection() ou createTopicConnection().

La classe qui implémente cette interface se charge du dialogue avec le broker de message.

La méthode start() permet de démarrer la connexion.

Exemple :

```
connection.start();
```

La méthode stop() permet de suspendre temporairement la connexion.

La méthode close() permet de fermer la connexion.

Remarque : il est important de fermer explicitement la connexion lorsqu'elle devient inutile en utilisant méthode close().

50.3.3. L'interface Session

Elle représente un contexte transactionnel de réception et d'émission pour une connexion donnée.

C'est d'ailleurs à partir d'un objet de type Connection que l'on créé une ou plusieurs sessions.

La session est mono thread : si l'application utilise plusieurs threads qui échangent des messages, il faut définir une session pour chaque thread.

C'est à partir d'un objet session que l'on créé des messages et des objets pour les envoyer et les recevoir.

Comme pour la connexion, la création d'un objet de type Session dépend du mode de fonctionnement. L'interface Session possède deux interfaces filles :

* l'interface QueueSession pour le mode point à point
* l'interface TopicSession pour le mode publication/abonnement

Pour obtenir l'un ou l'autre, il faut utiliser un objet Connection correspondant de type QueueConnection ou TopicConnection avec la méthode correspondante : createQueueSession() ou createTopicSession().

Ces deux méthodes demandent deux paramètres : un booléen qui indique si la session gère une transaction, et une constante qui précise le mode d'accusé de réception des messages.

Les messages sont considérés comme traités par le MOM à la réception d'un acquittement de réception. Celui ci est fourni au MOM selon le mode utilisé. Il existe trois modes d'accusés de réception (trois constantes sont définies dans l'interface Session) :

* AUTO_ACKNOWLEDGE : l'accusé de réception est automatique, le MOM reçoit l'acquittement à la réception du message que ce dernier soit traité ou non par l'application
* CLIENT_ACKNOWLEDGE : le MOM reçoit explicitement l'acquittement de la part de l'application, c'est le client qui envoi l'accusé grâce à l'appel de la méthode acknowledge() du message
* DUPS_OK_ACKNOWLEDGE : ce mode permet de dupliquer un message

L'interface Session définit plusieurs méthodes dont les principales sont :

Méthode	Rôle
void close()	Fermer la session

void commit()	Valider la transaction
XXX createXXX()	Créer un Message dont le type est XXX
void rollback()	Invalider la transaction

50.3.4. Les messages

Les messages sont encapsulés dans un objet de type javax.jms.Message : ils doivent obligatoirement implémenter l'interface Message ou l'une de ces sous classes.

Un message est constitué de trois parties :

- L'en-tête (header) : contient des données techniques
- Les propriétés (properties) : contient de données fonctionnelles
- Le corps du message (body) : contient les données du message

L'interface Session propose plusieurs méthodes createXXXMessage() pour créer des messages contenant des données au format XXX.

Il existe aussi pour chaque format des interfaces filles de l'interface Message :

- BytesMessage : message composé d'octets
- MapMessage : message composé de paire clé/valeur
- ObjectMessage : message contenant un objet sérialisé
- StreamMessage : message issu d'un flux
- TextMessage : message contenant du texte

50.3.4.1. L'en tête

Cette partie du message contient un certain nombre de champs prédéfinis qui contiennent des données pour identifier et acheminer le message.

La plupart de ces données sont renseignées lors de l'appel à la méthode send() ou publish().

L'en-tête contient des données standardisées dont le nom commence par JMS (JMSDestination, JMSDeliveryMode, JMSExpiration, JMSPriority, JMSMessageID, JMSTimestamp, JMSRedelivered, JMSCorrelationID, JMSReplyTo et JMSType)

Les champs les plus importants sont :

Nom	Rôle
JMSMessageID	Identifiant unique du message
JMSDestination	File d'attente ou topic destinataire du message
JMSCorrelationID	Utilisé pour synchroniser de façon applicative deux messages de la forme requête/réponse. Dans ce cas, dans le message réponse, ce champ contient le messageID du message requête

Les propriétés contiennent des données fonctionnelles sous la forme de paire clé/valeur. Certaines propriétés peuvent aussi être positionnées par l'implémentation du provider. Leur nom commence par JMS_ suivi du nom du provider.

50.3.4.2. Les propriétés

Ce sont des champs supplémentaires : certains sont définis par JMS mais il est possible d'ajouter ces propres champs.

Cette partie du message est optionnelle.

Elles permettent de définir des données qui seront utilisées pour fournir des données supplémentaires ou pour filtrer le message.

50.3.4.3. Le corps du message

Il contient les données du message : il est formaté selon le type du message.

Cette partie du message est optionnelle.

Les messages peuvent être de plusieurs types, définis dans les interfaces suivantes :

type	Interface	Rôle
bytes	BytesMessage	échange d'octets
texte	TextMessage	échange de données texte (XML par exemple)
object	ObjectMessage	échange d'objets Java qui doivent être sérialisables
Map	MapMessage	échange de données sous la forme clé/valeur. La clé doit être une chaîne de caractères et la valeur de type primitive
Stream	StreamMessage	échange de données en provenance d'un flux

Il est possible de définir son propre type qui doit obligatoirement implémenter l'interface Message.

C'est un objet de type Session qui contient les méthodes nécessaires à la création d'un message selon son type.

Lors de la réception d'un message, celui ci est toujours de type Message : il faut effectuer un transtypage en fonction de son type en utilisant l'opérateur instanceof. A ce moment, il faut utiliser le getter correspondant pour obtenir les données.

Exemple :

```
Message message = ...

if (message instanceof TextMessage) {
   TextMessage textMessage = (TextMessage) message;
   System.out.println("message: " + textMessage.getText());
}
```

50.3.5. L'envoi de messages

L'interface MessageProducer est la super interface des interfaces qui définissent des méthodes pour l'envoi de messages.

Il existe deux interfaces filles selon le mode de fonctionnement pour envoyer un message : QueueSender et TopicPublisher.

Ces objets sont créés à partir d'un objet représentant la session :

- la méthode createSender() pour obtenir un objet de type QueueSender
- la méthode createPublisher() pour obtenir un objet de type TopicPublisher

Ces objets peuvent être liés à une entité physique par exemple une file d'attente particulière pour un objet de type QueueSender. Si ce n'est pas le cas, cette entité devra être précisée lors de l'envoie du message en utilisant une version surchargée de la méthode chargée de l'émission du message.

50.3.6. La réception de messages

L'interface MessageConsumer est la super interface des interfaces qui définissent des méthodes pour la réception de messages.

Il existe des interfaces selon le mode de fonctionnement pour recevoir un message QueueReceiver et TopicSubscriber.

La réception d'un message peut se faire avec deux modes :

- synchrone : dans ce cas, l'attente d'un message bloque l'éxecution du reste de code
- asynchrone : dans ce cas, un thread est lancé qui attend le message et appelle une méthode (callback) à son arrivée. L'exécution de l'application n'est pas bloquée.

L'interface MessageConsumer définit plusieurs méthodes sont les principales sont :

Méthode	Rôle
close()	Fermer l'objet qui reçoit les messages pour le rendre inactif
Message receive()	Attendre et retourner le message à son arrivée
Message receice(long)	Attendre durant le nombre de milliseconde précisé en paramètre et renvoie le message s'il arrive durant ce laps de temps
Message receiveNoWait()	Renvoyer un message sans attendre si il y en a un de présent
setMessageListener(MessageListener)	Associer un Listener pour traiter les messages de façon asynchrone

Pour obtenir un objet qui implémente l'interface QueueReceiver, il faut utiliser la méthode createReceiver() d'un objet de type QueueSession.

Pour obtenir un objet qui implémente l'interface TopicSubscriber, il faut utiliser la méthode createSubscriber() d'un objet de type TopicSession.

50.4. L'utilisation du mode point à point (queue)

50.4.1. La création d'une factory de connexion : QueueConnectionFactory

Un objet factory est un objet qui permet de retourner un objet pour se connecter au broker de messages.

Pour obtenir un objet de ce type, il faut soit instancier directement un tel objet soit faire appel à JNDI pour l'obtenir.

Exemple : avec MQSeries

```
String qManager = ...
String hostName = ...
String channel = ...

MQQueueConnectionFactory factory = new MQQueueConnectionFactory();
factory.setQueueManager(qManager);
factory.setHostName(hostName);
factory.setChannel(channel);
factory.setTransportType(JMSC.MQJMS_TP_CLIENT_MQ_TCPIP);
```

Il est cependant préférable de faire appel à JNDI pour obtenir un objet de type QueueConnectionFactory. Une instance de cet objet est stocké dans un annuaire par le broker et il suffit de se connecter à cet annuaire via JNDI pour obtenir l'instance de la fabrique.

50.4.2. L'interface QueueConnection

Cette interface hérite de l'interface Connection.

Pour obtenir un objet qui implémente cette interface, il faut utiliser un objet de type QueueConnectionFactory avec la méthode correspondante : createQueueConnection().

Exemple :

```
            QueueConnection connection = factory.createQueueConnection();
            connection.start();
```

L'interface QueueConnection définit plusieurs méthodes dont la principale est :

Méthode	Rôle
QueueSession createQueueSession(boolean, int)	Renvoyer un objet qui définit la session. Le booléen précise si la session gère une transaction. L'entier précise le mode d'accusé de réception.

50.4.3. La session : l'interface QueueSession

Elle hérite de l'interface Session.

Pour obtenir un objet qui implémente cette interface, il faut utiliser la méthode createQueueSession() d'un objet de type QueueConnection.

Exemple :

```
QueueSession session = connection.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
```

L'interface QueueSession définit plusieurs méthodes dont les principales sont :

Méthode	Rôle
QueueReceiver createQueueReceiver(Queue)	Renvoyer un objet qui définit une file d'attente de réception
QueueSender createQueueSender(Queue)	Renvoyer un objet qui définit une file d'attente d'émission

50.4.4. L'interface Queue

Un objet qui implémente cette interface encapsule une file d'attente particulière.

Pour obtenir un objet qui implémente cette interface, il faut utiliser la méthode createQueue() d'un objet de type QueueSession.

Exemple avec MQseries :

```
   Queue fileEnvoi =
      session.createQueue("queue:///file.out"?expiry=0&persistence=1&targetClient=1");
```

50.4.5. La création d'un message

Pour créer un message, il faut utiliser une méthode createXXXMessage() d'un objet QueueSession ou XXX représente le type du message.

Exemple :

```
String message = "bonjour";
TextMessage textMessage = session.createTextMessage();
textMessage.setText(message);
```

50.4.6. L'envoi de messages : l'interface QueueSender

Cette interface hérite de l'interface MessageProducer.

Pour obtenir un objet qui implémente cette interface, il faut utiliser la méthode createQueueSender() d'un objet de type QueueSession.

Exemple :

```
QueueSender queueSender = session.createSender(fileEnvoi);
```

Il est possible de fournir un objet de type Queue qui représente la file d'attente : dans ce cas, l'objet QueueSender est lié à cette file d'attente. Si l'on ne précise pas de file d'attente (null fourni en paramètre), il faudra obligatoirement utiliser une version surchargée de la méthode send() lors de l'envoi pour préciser la file d'attente à utiliser.

Avec un objet de type QueueSender, la méthode send() permet l'envoie d'un message dans la file d'attente. Cette méthode possède plusieurs surcharges :

Méthode	Rôle
void send(Message)	Envoyer le message dans la file d'attente définit dans l'objet de type QueueSender
void send(Queue, Message)	Envoyer le message dans la file d'attente fournie en paramètre

Exemple :

```
queueSender.send(textMessage);
```

50.4.7. La réception de messages : l'interface QueueReceiver

Cette interface hérite de l'interface MessageConsummer.

Pour obtenir un objet qui implémente cette interface, il faut utiliser la méthode createQueueReceiver() à partir d'un objet de type QueueSession.

Exemple :

```
QueueReceiver queueReceiver = session.createReceiver(fileReception);
```

Il est possible de fournir un objet de type Queue qui représente la file d'attente : dans ce cas, l'objet QueueSender est lié à cette file d'attente. Si l'on ne précise pas de file d'attente (null fourni en paramètre), dans ce cas, il faudra obligatoirement

utiliser une version surchargée de la méthode receive() lors de l'envoi pour préciser la file d'attente.

Cette interface ne définit qu'une seule méthode supplémentaire :

Méthode	Rôle
Queue getQueue()	Renvoyer la file d'attente associée à l'objet

La réception de messages peut se faire dans le mode synchrone ou asynchrone.

50.4.7.1. La réception dans le mode synchrone

Dans ce mode, le programme est interrompu jusqu'à l'arrivée d'un nouveau message. Il faut utiliser la méthode receive() héritée de l'interface MessageConsumer. Il existe plusieurs méthodes et surcharges de ces méthodes qui permettent de répondre à plusieurs utilisations :

- receiveNoWait() : renvoi un message présent sans attendre
- receive(long) : renvoi un message qui arrive durant le temps fourni en paramètre
- receive() : renvoi le message dès qu'il arrive

Exemple :

```
          Message message = null;
          message = queueReceiver.receive(10000);
```

50.4.7.2. La réception dans le mode asynchrone

Dans ce mode, le programme n'est pas interrompu mais un objet écouteur va être enregistré auprès de l'objet de type QueueReceiver. Cet objet qui implémente l'interface MessageListener va être utilisé comme gestionnaire d'événements lors de l'arrivée d'un nouveau message.

L'interface MessageListener ne définit qu'une seule méthode qui reçoit en paramètre le message : onMessage(). C'est cette méthode qui sera appelée lors de la réception d'un message.

50.4.7.3. La sélection de messages

Une version surchargée de la méthode createReceiver() d'un objet de type QueueSession permet de préciser dans ces paramètres une chaîne de caractères qui va servir de filtre sur les messages à recevoir.

Dans ce cas, le filtre est effectué par le broker de message plutôt que par le programme.

Cette chaîne de caractères contient une expression qui doit avoir une syntaxe proche d'une condition SQL. Les critères de la sélection doivent porter sur des champs inclus dans l'en tête ou dans les propriétés du message. Il n'est pas possible d'utiliser des données du corps du message pour effectuer le filtre.

Exemple : envoi d'un message requête et attente de sa réponse. Dans ce cas, le champ JMSCorrelationID du message réponse contient le JMSMessageID du message requete

```
           String messageEnvoie = "bonjour";
           TextMessage textMessage = session.createTextMessage();
           textMessage.setText(messageEnvoie);
           queueSender.send(textMessage);
           int correlId = textMessage.getJMSMessageID();
           QueueReceiver queueReceiver = session.createReceiver(
           fileEnvoie, "JMSCorrelationID = '" + correlId +"'");
```

```
            Message message = null;
            message = queueReceiver.receive(10000);
```

50.5. L'utilisation du mode publication/abonnement (publish/subscribe)

50.5.1. La création d'une factory de connexion : TopicConnectionFactory

Un objet factory est un objet qui permet de retourner un objet pour se connecter au broker de messages.

Pour obtenir un objet de ce type, il faut soit instancier directement un tel objet soit faire appel à JNDI pour l'obtenir.

50.5.2. L'interface TopicConnection

Cette interface hérite de l'interface Connection.

Pour obtenir un objet qui implémente cette interface, il faut utiliser un objet factory correspondant de type TopicConnectionFactory avec la méthode correspondante : createTopicConnection().

Exemple :

```
            TopicConnection connection = factory.createTopicConnection();
            connection.start();
```

L'interface TopicConnection définit plusieurs méthodes dont la principale est :

Méthode	Rôle
TopicSession createTopicSession(boolean, int)	Renvoyer un objet qui définit la session. Le booléen précise si la session gère une transaction. L'entier précise le mode d'accusé de réception.

50.5.3. La session : l'interface TopicSession

Elle hérite de l'interface Session.

Pour obtenir un objet qui implémente cette interface, il faut utiliser la méthode createTopicSession() d'un objet connexion de type TopicConnection.

Exemple :

```
      TopicSession session = connection.createTopicSession(false,Session.AUTO_ACKNOWLEDGE);
```

L'interface TopicSession définit plusieurs méthodes dont les principales sont :

Méthode	Rôle
TopicSubscriber createSubscriber(Topic)	Renvoyer un objet qui permet l'envoie de messages dans un topic
TopicPublisher createPublisher(Topic)	Renvoyer un objet qui permet la réception de messages dans un topic
Topic createTopic(String)	Créer un topic correspondant à la désignation fournie en paramètre

50.5.4. L'interface Topic

Un objet qui implémente cette interface encapsule un sujet.

Pour obtenir un objet qui implémente cette interface, il faut utiliser la méthode createTopic() d'un objet de type TopicSession.

50.5.5. La création d'un message

Pour créer un message, il faut utiliser une méthode createXXXMessage() d'un objet TopicSession ou XXX représente le type du message.

Exemple :

```
String message = "bonjour";
TextMessage textMessage = session.createTextMessage();
textMessage.setText(message);
```

50.5.6. L'émission de messages : l'interface TopicPublisher

Cette interface hérite de l'interface MessageProducer.

Avec un objet de type TopicPublisher, la méthode publish() permet l'envoie du message. Cette méthode possède plusieurs surcharges :

Méthode	Rôle
void publish(Message)	Envoyer le message dans le topic défini dans l'objet de type TopicPublisher
void publish(Topic, Message)	Envoyer le message dans le topic fourni en paramètre

50.5.7. La réception de messages : l'interface TopicSubscriber

Cette interface hérite de l'interface MessageProducer.

Pour obtenir un objet qui implémente de cette interface, il faut utiliser la méthode createSubscriber() à partir d'un objet de type TopicSession.

Exemple :

```
TopicSubscriber topicSubscriber = session.createSubscriber(topic);
```

Il est possible de fournir un objet de type Topic qui représente le topic : dans ce cas, l'objet TopicSubscriber est lié à ce topic. Si l'on ne précise pas de topic (null fourni en paramètre), il faudra obligatoirement utilisée une version surchargé de la méthode receive() lors de l'envoi pour préciser le topic.

Cette interface ne définit qu'une seule méthode supplémentaire :

Méthode	Rôle
Topic getTopic()	Renvoyer le topic associé à l'objet

50.6. La gestion des erreurs

Les erreurs d'exécution liées à l'utilisation de JMS sont rapportées sous la forme d'exception. La plupart des méthodes des objets JMS peuvent lever une exception de type JMSException.

Lors de l'utilisation de message asynchrone, il est possible d'enregistrer un listener de type ExceptionListener. Une instance de ce listener redéfini la méthode onException() qui attend en paramètre une instance de type JMSException

50.6.1. Les exceptions de JMS

Plusieurs exceptions sont définies par l'API JMS. La classe mère de toute ces exceptions est la classe JMSException.

Les exceptions définies sont : IllegalStateException, InvalidClientIDException, InvalidDestinationException, InvalidSelectorException, JMSSecurityException, MessageEOFException, MessageFormatException, MessageNotReadableException, MessageNotWriteableException, ResourceAllocationException, TransactionInProgressException,TransactionRolledBackException

La méthode getErrorCode() permet d'obtenir le code erreur spécifique du produit sous forme de chaîne de caractères.

50.6.2. L'interface ExceptionListener

Ce listener permet d'être informé des exceptions levées par le provider JMS (exemple : arrêt du serveur, problème réseau, ..).

Cette interface définit la méthode OnException() qui doit être implémentée pour contenir les traitements en cas d'erreur.

Exemple :

```
import javax.jms.ExceptionListener;
import javax.jms.JMSException;

public class MonExceptionListener implements ExceptionListener {

    public void onException(JMSException jmse) {
        jmse.printStackTrace(System.err);
    }
}
```

50.7. JMS 1.1

La version 1.1 de JMS propose une utilisation de l'API indépendamment du domaine utilisé et ainsi d'unifier l'API pour utiliser le mode d'utilisation point à point et publication/souscription. Avec cette version, l'utilisation d'un mode ou l'autre ne nécessite plus l'utilisation d'interfaces spécifiques au mode utilisé, ce qui rend l'API plus simple à utiliser.

JMS 1.1 contient toujours toutes interfaces dépendants du domaine utilisé mais propose aussi enrichissement des interfaces communes pour permettent leur utilisation indépendamment du domaine utilisé.

La version 1.0.2 de l'API définit trois familles d'interfaces : commune, Queue et Topic. Pour chaque mode, une interface spécifique est définie pour la fabrique, la connexion, la session, la production et la consommation de messages.

Interfaces communes	Interfaces point à point	Interfaces publication/souscription
ConnectionFactory	QueueConnectionFactory	TopicConnectionFactory
Connection	QueueConnection	TopicConnection

Session	QueueSession	TopicSession
Destination	Queue	Topic
MessageProducer	QueueSender	TopicPublisher
MessageConsumer	QueueReceiver QueueBrowser	TopicSubscriber

JMS propose aussi 9 interfaces supplémentaires pour le support des transactions distribuées avec XA.

Avec JMS 1.1 il est possible de généraliser l'utilisation des interfaces communes que ce soit pour une utilisation dans le mode point à point ou publication/souscription. Ces interfaces communes ont été enrichies pour rendre les interfaces filles polymorphiques. Par exemple, l'interface MessageProducer possède une méthode send() pour permettre à un client d'envoyer un message dans un mode ou un autre. Ainsi l'interface MessageProducer permet de réaliser les actions des interfaces QueueSender et TopicPublisher.

Le code devient donc plus simple, plus générique et plus réutilisable.

Ceci permet de rendre le code indépendant de la solution utilisée : l'utilisation d'une instance de type Destination se fait indépendamment que celle soit une Queue ou un Topic.

Ceci permet aussi dans une même session d'utiliser une queue et un topic simultanément alors que les versions précédentes de JMS, il était nécessaire de définir deux sessions.

JMS 1.1 permet l'utilisation de destinations qui n'ont pas besoin de savoir si celle-ci concernent un Queue ou un Topic : le code écrit peut utiliser indifféremment l'un ou l'autre.

La version 1.1 a été diffusée en avril 2002. Cette version est intégrée à J2EE 1.4 : c'est un pré requis pour la version 2.1 des EJB.

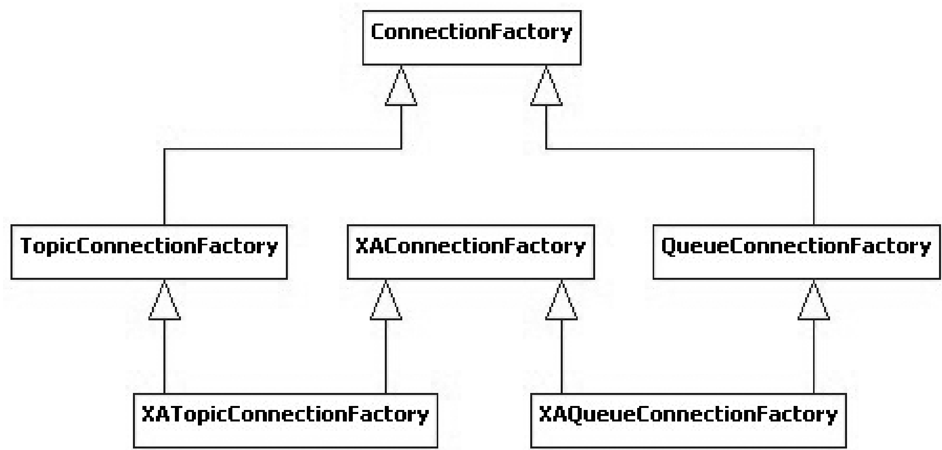
50.7.1. L'utilisation de l'API JMS 1.0 et 1.1

Point à point	Publication/souscription	Point à point ou Publication/souscription
JMS 1.0 ou 1.1	JMS 1.0 ou 1.1	JMS 1.1 uniquement
Obtenir une instance de la fabrique de type connectionFactory		
Obtenir une instance de QueueConnectionFactory à partir de JNDI	Obtenir une instance de TopicConnectionFactory à partir de JNDI	Obtenir une instance de ConnectionFactory à partir de JNDI
Créer une instance de Connection		
Appel de la méthode createQueueConnection() de la fabrique	Appel de la méthode createTopicConnection() de la fabrique	Appel de la méthode createConnection() de la fabrique
Créer une instance de Session		
Appel de la méthode createQueueSession() de la connexion	Appel de la méthode createTopicSession() de la connexion	Appel de la méthode createSession de la connexion
Créer une instance de MessageProducer		
Créer une instance de QueueSender en utilisant la méthode createSender() de la session	Créer une instance de TopicPublisher en utilisant la méthode createPublisher() de la session	Créer une instance de TopicPublisher en la méthode createProducer() de la session
Envoyer un message		

Utiliser la méthode send() de la classe QueueSender	Utiliser la méthode publish() de la classe TopicPublisher	Utiliser la méthode send() de la classe MessageProducer
Créer une instance de MessageConsumer		
Créer une instance de la classe QueueReceiver en utilisant la méthode createReceiver() de la session	Créer une instance de la classe QueueReceiver en utilisant la méthode createReceiver() de la session	Créer une instance de la classe MessageConsumer en utilisant la méthode createConsumer() de la session
Recevoir un message de façon synchrone		
Appel de la méthode receive() de l'instance de QueueReceiver	Appel de la méthode receive() de l'instance de TopicSubscriber	Appel de la méthode receive() de l'instance de MessageConsumer
Recevoir un message de façon asynchrone		
Implémenter l'interface MessageListerner et l'enregistrer avec la méthode setMessageListener() de la classe QueueReceiver	Implémenter l'interface MessageListerner et l'enregistrer avec la méthode setMessageListener() de la classe TopicSubscriber	Implémenter l'interface MessageListerner et l'enregistrer avec la méthode setMessageListener() de la classe MessageConsumer

50.7.2. L'interface ConnectionFactory

Un objet de type ConnectionFactory est une fabrique qui permet d'obtenir une instance de l'interface Connection. Il faut interroger un annuaire JNDI pour obtenir une instance de cette fabrique.

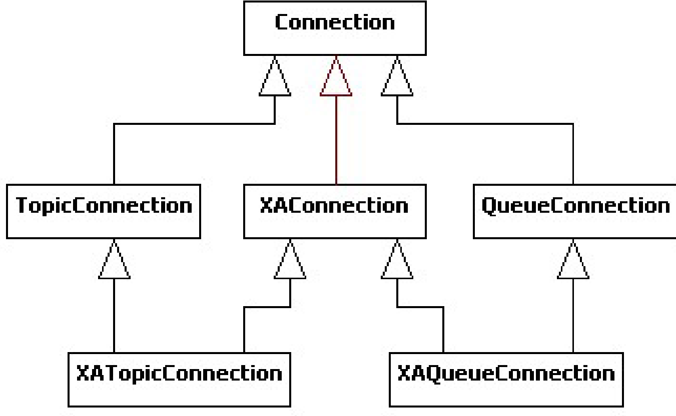


Avec JMS 1.1, il est maintenant possible d'utiliser une instance de ConnectionFactory directement : il n'est plus nécessaire comme dans les versions précédentes d'utiliser une fabrique dédiée à l'utilisation de Queue ou de Topic.

Remarque : Avec certaines implémentations, il est possible de créer manuellement une instance de ConnectionFactory mais delà nécessite de faire appel des objets spécifiques à l'implémentation ce qui rend le code dépendant de l'implémentation et donc moins portable.

50.7.3. L'interface Connection

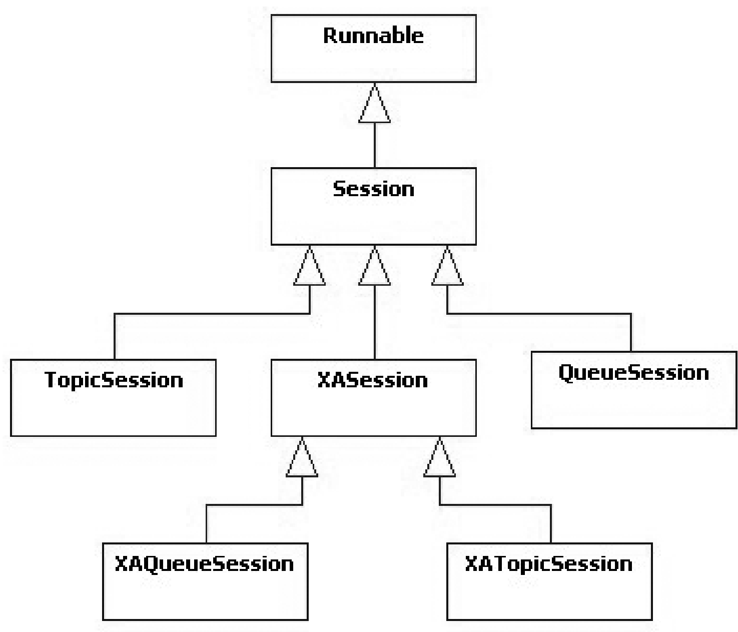
L'interface Connection permet de se connecter au serveur JMS. Avec JMS1.1, pour obtenir une instance du type Connection, il faut utiliser une des surcharges de la méthode createConnection() de l'interface ConnectionFactory.



La méthode start() de l'interface Connection permet de démarrer la connexion.

50.7.4. L'interface Session

Une session est une fabrique de messages et elle encapsule un contexte dans lequel les messages sont produits et consommés.



Une session JMS permet de créer les objets de type MessageProducer, MessageConsumer et Message.

Pour obtenir une instance de l'interface Session, il utiliser la méthode createSession(). Depuis JMS 1.1, cette méthode est disponible dans l'interface Connection.

Depuis JMS 1.1, de nouvelles méthodes ont été ajoutées à l'interface Session :

Méthode	Rôle
MessageProducer createProducer()	
MessageConsumer createConsumer()	
Queue createQueue()	

Topic createTopic()	
TopicSubscriber createDurableSubscriber()	
QueueBrowser createBrowser()	
TemporaryTopic createTemporaryTopic()	
TemporaryQueue createTemporaryQueue()	
void unsubscribe()	

Pour obtenir une session, il faut utiliser la méthode createSession() de l'interface Connection.

Cette méthode attend deux paramètres :

- Un booléen qui précise si la session est transactionnelle (true) ou non (false)
- Un entier qui précise le mode d'acquittement de la réception d'un message (Session.AUTO_ACKNOWLEDGE, Session.CLIENT_ACKNOWLEDGE, ou Session.DUPS_OK_ACKNOWLEDGE)

Une connexion JMS est Thread safe par contre la session JMS ne l'est pas : il faut donc utiliser une session par thread.

Une session JMS peut être transactionnelle en passant la valeur true au paramètre transacted des methodes createSession(), createQueueSession() ou createTopicSession().
Pour valider la transaction, il faut utiliser la méthode commit() de l'interface Session.

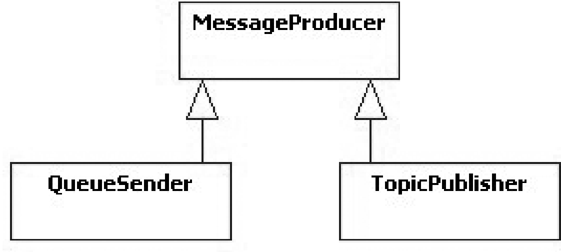
50.7.5. L'interface Destination

L'interface Destination est la super interface des interfaces Queue et Topic.

Avec JMS 1.1, il est préférable d'utiliser cette interface plutôt que d'utiliser une interface dédiée au domaine utilisé.

50.7.6. L'interface MessageProducer

L'interface MessageProducer permet d'envoyer un message vers une destination indépendamment du domaine utilisé (queue ou topic)



Avec JMS 1.1, une instance est obtenue en utilisant la méthode createProducer() de l'interface Session avec en paramètre la destination. Il est aussi possible de créer un MessageProducer sans préciser la destination. Dans ce cas, cette dernière devra être précisée lors de l'envoi du message.

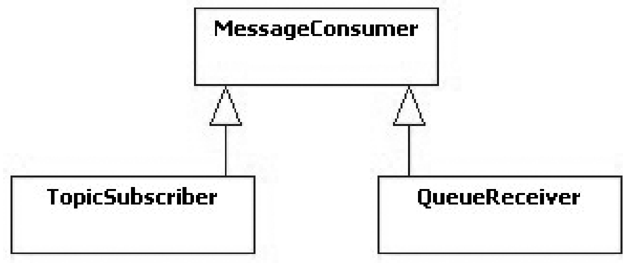
Depuis JMS 1.1, il est possible d'utiliser une instance de cette interface pour produire des messages. Avec JMS 1.0, il était nécessaire d'utiliser TopicPublisher ou QueueSender.

Depuis JMS 1.1, de nouvelles méthodes ont été ajoutées à l'interface MessageProducer notamment la méthode getDestination() et plusieurs surcharges de la méthode send()

Les surcharges de la méthode send() de l'interface MessageProducer permet d'envoyer un message fourni en paramètre.

50.7.7. L'interface MessageConsumer

L'interface MessageConsumer permet la réception de messages d'une destination. Une instance est obtenue en utilisant la méthode createConsumer() de l'interface Session. Cette méthode attend en paramètre une destination.



50.7.7.1. La réception synchrone de messages

La méthode receive() de l'interface MessageConsumer permet d'attendre l'arrivée d'un nouveau message en bloquant le reste de l'application. Une version surchargée attend en paramètre un nombre de millisecondes durant laquelle se fera l'attente au maximum

La méthode receiveNoWait() permet de recevoir un éventuel nouveau message sans attendre.

Un message reçu est retourné par ces méthodes sous la forme d'un objet de type Message. Pour traiter le message, il faut caster ce résultat en fonction du type réel de l'objet.

50.7.7.2. La réception asynchrone de messages

La méthode receive() de la classe MessageConsumer permet de recevoir un message de façon synchrone. Lors de l'appel à cette méthode un message est obtenu ou non.

L'arrivée d'un message est cependant rarement prédictible et surtout ne doit pas bloquer l'exécution de l'application. Il est alors préférable de définir un listener et de l'enregistrer pour qu'il soit automatiquement exécuté à l'arrivée d'un message.

L'interface MessageListener permet de définir un listener pour la réception asynchrone de messages. Elle ne définit que la méthode onMessage() qui sera appelée lors de chaque réception d'un nouveau message de la destination.

La méthode onMessage() possède un paramètre de type Message encapsulant le message reçu. Il faut redéfinir cette méthode pour qu'elle exécute les traitements à réaliser sur les messages.

Le listener s'enregistre en utilisant la méthode setMessageListener() de la classe MessageConsumer().

Exemple :

```
messageConsumer.setMessageListener(listener);
```

Remarque : il est important d'enregistrer le listener après que la connexion au serveur soit réalisée (appel de la méthode start() de la Connection).

50.7.8. Le filtrage des messages

Il est possible de filtrer les messages reçus d'une destination au moyen d'un sélecteur (selector). Les fonctionnalités utilisables correspondent à un petit sous ensemble de l'ensemble des fonctionnalités de SQL.

Le filtre ne peut s'appliquer que sur certaines données de l'en-tête : JMSDeliveryMode, JMSPriority, JMSMessageID, JMSCorrelationID, JMSType et JMSTimestamp

Le filtre peut aussi utiliser toutes les propriétés personnelles du message.

Exemple :

```
JMSPriority < 10
```

Lors de l'instanciation d'un objet de type MessageConsumer, il est possible de préciser le filtre des messages à recevoir sous la forme d'une chaîne de caractères. Cette chaîne est une expression qui précise le filtre à appliquer sur les messages pour ne recevoir que ceux qui satisfassent la condition précisée dans le filtre. Cette expression est nommée selector.

Exemple :

```
        messageConsumer consumer = session.createConsumer(destination, "maPropriete = '1234' ") ;
```

Il est possible de définir ces propres propriétés et de les utiliser dans le filtre. Le nom de ces propriétés doit impérativement respecter les spécifications de JMS (par exemple, le nom ne peut pas commencer par JMSX ou JMS_).

La valeur d'une propriété peut être de type boolean, byte, short, int, long, float, double ou String

Les valeurs des propriétés sont précisées avant l'envoi du message et ne peuvent plus être modifiées après l'envoi du message.

Les spécifications JMS ne précisent pas de règle pour l'utilisation d'une donnée sous la forme d'une propriété ou dans le corps du message. Il est cependant conseillée de réservé l'utilisation des propriétés pour des besoins spécifiques (filtre de messages par exemple).

Les filtres permettent à un client de ne recevoir que les messages dont les données de l'en-tête respectent le filtre précisé. Il n'est pas possible d'utiliser dans le filtre les données du corps du message.

Les messages retenus sont ceux dont l'évaluation de l'expression avec les valeurs de l'en-tête du message vaut true.

Le filtre ne peut pas être changé en cours d'exécution.

50.7.8.1. La définition du filtre

Le filtre, nommé selector est une chaîne de caractères définissant une expression dont la syntaxe est un sous ensemble des expressions conditionnelles de la norme SQL 92.

Par défaut, le filtre est évalué de gauche à droite mais l'usage de parenthèses peut être mis en oeuvre pour modifier cet ordre.

Un selector peut contenir :

des séparateurs	espaces, tabulations, retour chariot, ...
des littéraux	des chaînes de caractères encodés en Unicode et entourés par de simples quotes des numériques entiers correspondant au type Java long des numériques flottants correspondant au type Java double des booléens qui peuvent avoir les valeurs true ou false

des identifiants	leur nom doit respecter ceux des identifiants Java et ne doivent pas correspondre à des mots clés (true, false, null, not, and, or, ...) ils ne doivent pas commencer par JMSX ou JMS_ ils sont sensibles à la casse ils ne peuvent pas correspondre aux propriétés d'en-tête prédéfinis : JMSDeliveryMode, JMSPriority, JMSMessageID, JMSTimestamp, JMSCorrelationID ou JMSType
des parenthèses	pour modifier l'ordre d'évaluation de l'expression
des expressions	arithmétiques conditionnelles
des opérateurs logiques	NOT, AND, OR
des opérateurs de comparaisons	=, >, >=, <, <=, <> (seuls = et <> sont utilisables avec des booléens et des chaînes de caractères
des opérateurs arithmétiques	+, -, *, /
l'opérateur between	exemple : valeur between 5 and 9 est équivalent à valeur >= 5 et valeur <= 9, valeur not between 5 and 9 est équivalent à valeur < 5 ou valeur > 9
l'opérateur in	permet la comparaison parmi plusieurs chaînes de caractères (exemple : valeur in («aa','bb, «cc') est équivalant à (valeur = «aa') ou (valeur = «bb') ou (valeur = «cc'))
l'opérateur like	permet la comparaison par rapport à un motif : dans ce motif le caractère _ désigne un caractère quelconque, le caractère % désigne zéro ou plusieurs caractères, le caractères \ permet de déspécialiser les deux précédents caractères
L'opérateur is null	permet de comparer la valeur null pour une propriété ou si une propriété n'est pas définie

50.7.9. Des exemples de mise en oeuvre

Exemple : envoie d'un message dans une queue

```
package com.jmdoudoux.test.openjms;

import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.MessageProducer;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class TestOpenJMS1 {

  public static void main(final String[] args) {
    Context context = null;
    ConnectionFactory factory = null;
    Connection connection = null;
    Destination destination = null;
    Session session = null;
    MessageProducer sender = null;
    try {
      context = new InitialContext();
      factory = (ConnectionFactory) context.lookup("ConnectionFactory");
      destination = (Destination) context.lookup("queue1");
      connection = factory.createConnection();
      session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
      sender = session.createProducer(destination);
      connection.start();

      final TextMessage message = session.createTextMessage();
```

```
      message.setText("Mon message");
      sender.send(message);
      System.out.println("Message envoye= " + message.getText());
    } catch (Exception e) {
      e.printStackTrace();
    } finally {
      if (context != null) {
        try {
          context.close();
        } catch (Exception e) {
          e.printStackTrace();
        }
      }

      if (connection != null) {
        try {
          connection.close();
        } catch (Exception e) {
          e.printStackTrace();
        }
      }
    }
  }

}
```

Pour exécuter correctement l'application il faut qu'un broker de messages JMS soit installé et configuré. Il suffit alors de fournir les paramètres de connexion à ce serveur.

Exemple : le fichier jndi.properties avec OpenJMS

```
java.naming.provider.url=tcp://localhost:3035
java.naming.factory.initial=org.exolab.jms.jndi.InitialContextFactory
java.naming.security.principal=admin
java.naming.security.credentials=openjms
```

Résultat :

```
Message envoye= Mon message
```

Grâce à la version 1.1 de JMS, pour envoyer un message dans le topic1, il suffit simplement le remplacer le nom JNDI de la destination

Exemple :

```
...
      destination = (Destination) context.lookup("topic1");
...
```

Exemple : lecture d'un message dans une file d'attente

```
package com.jmdoudoux.test.openjms;

import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageConsumer;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class TestOpenJMS2 {
```

```
  public static void main(String[] args) {
    Context context = null;
    ConnectionFactory factory = null;
    Connection connection = null;
    Destination destination = null;
    Session session = null;
    MessageConsumer receiver = null;

    try {
      context = new InitialContext();
      factory = (ConnectionFactory) context.lookup("ConnectionFactory");
      destination = (Destination) context.lookup("queue1");
      connection = factory.createConnection();
      session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
      receiver = session.createConsumer(destination);
      connection.start();

        Message message = receiver.receive();
        if (message instanceof TextMessage) {
          TextMessage text = (TextMessage) message;
          System.out.println("message recu= " + text.getText());
        } else if (message != null) {
          System.out.println("Aucun message dans la file");
        }
    } catch (Exception e) {
      e.printStackTrace();
    } finally {
      if (context != null) {
        try {
          context.close();
        } catch (NamingException e) {
          e.printStackTrace();
        }
      }

      if (connection != null) {
        try {
          connection.close();
        } catch (JMSException e) {
          e.printStackTrace();
        }
      }
    }
  }
}
```

50.8. Les ressources relatives à JMS

Le site de JMS.

La Documentation de l'API JMS en version 1.0.2b et 1.1.

Pour mettre en oeuvre JMS, il faut une implémentation de l'API fournie soit par un serveur d'application soit par une implémentation autonome.

Sun Java System Message Queue, l'implémentation JMS de Sun

OpenJMS est une implémentation Open Source des spécifications JMS

JORAM est une implémentation open source des spécifications JMS par le consortium ObjectWeb

51. Les EJB (Entreprise Java Bean)

Les Entreprise Java Bean ou EJB sont des composants serveurs donc non visuels qui respectent les spécifications d'un modèle édité par Sun. Ces spécifications définissent une architecture, un environnement d'exécution et un ensemble d'API.

Le respect de ces spécifications permet d'utiliser les EJB de façon indépendante du serveur d'applications J2EE dans lequel ils s'exécutent, du moment où le code de mise en oeuvre des EJB n'utilise pas d'extensions proposées par un serveur d'applications particulier.

Le but des EJB est de faciliter la création d'applications distribuées pour les entreprises.

Une des principales caractéristiques des EJB est de permettre aux développeurs de se concentrer sur les traitements orientés métiers car les EJB et l'environnement dans lequel ils s'exécutent prennent en charge un certain nombre de traitements tel que la gestion des transactions, la persistance des données, la sécurité, ...

Physiquement, un EJB est un ensemble d'au moins deux interfaces et une classe regroupées dans un module contenant un descripteur de déploiement particulier.

Pour obtenir des informations complémentaires sur les EJB, il est possible de consulter le site de Sun : java.sun.com/products/ejb

Il existe plusieurs versions des spécifications des E.J.B. :

- 1.0 :
- 1.1 :
- 2.0 :
- 2.1 :
- 3.0 :

Remarque : dans ce chapitre, le mot bean sera utilisé comme synonyme d'EJB. Ce chapitre couvre essentiellement la version 2.x des EJB.

Ce chapitre contient plusieurs sections :

- ♦ La présentation des EJB
- ♦ Les EJB session
- ♦ Les EJB entité
- ♦ Les outils pour développer et mettre oeuvre des EJB
- ♦ Le déploiement des EJB
- ♦ L'appel d'un EJB par un client
- ♦ Les EJB orientés messages

La suite de ce chapitre sera développée dans une version future de ce document

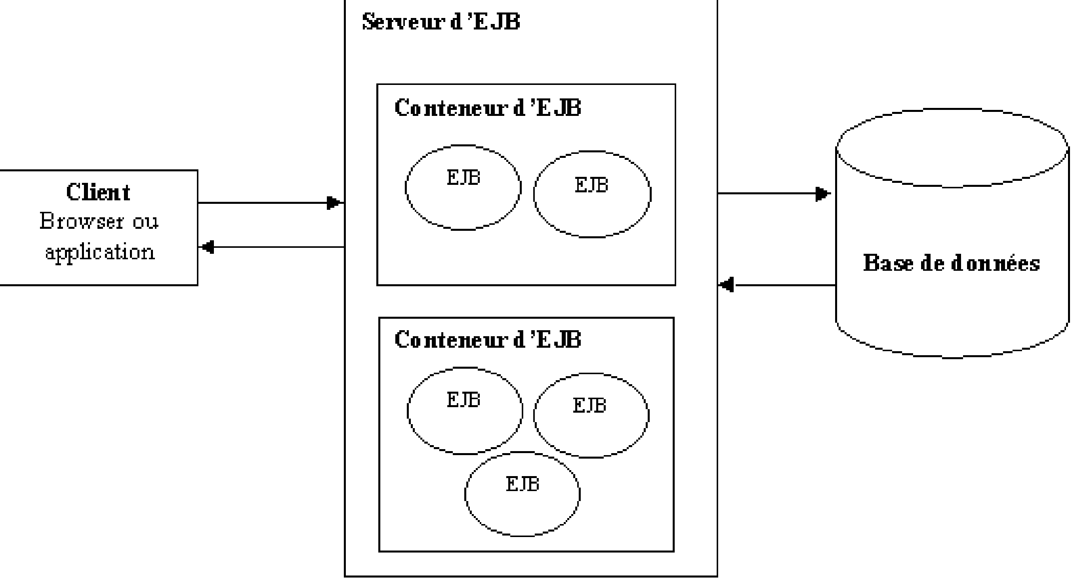
51.1. La présentation des EJB

Les EJB sont des composants et en tant que tels, ils possèdent certaines caractéristiques comme la réutilisabilité, la possibilité de s'assembler pour construire une application, etc ... Les EJB et les beans n'ont en commun que d'être des composants. Les java beans sont des composants qui peuvent être utilisés dans toutes les circonstances. Les EJB doivent obligatoirement s'exécuter dans un environnement serveur dédié.

Les EJB sont parfaitement adaptés pour être intégrés dans une architecture trois tiers ou plus. Dans une telle architecture, chaque tiers assure une fonction particulière :

- le client « léger » assure la saisie et l'affichage des données
- sur le serveur, les objets métiers contiennent les traitements. Les EJB sont spécialement conçus pour constituer de telles entités.
- une base de données assure la persistance des informations

Les EJB s'exécutent dans un environnement particulier : le serveur d'EJB. Celui ci fournit un ensemble de fonctionnalités utilisées par un ou plusieurs conteneurs d'EJB qui constituent le serveur d'EJB. En réalité, c'est dans un conteneur que s'exécute un EJB et il lui est impossible de s'exécuter en dehors.



Le conteneur d'EJB propose un certain nombre de services qui assurent la gestion :

- du cycle de vie du bean
- de l'accès au bean
- de la sécurité d'accès
- des accès concurrents
- des transactions

Les entités externes au serveur qui appellent un EJB ne communiquent pas directement avec celui ci. Les accès au EJB par un client se font obligatoirement via le conteneur. Un objet héritant de la classe EJBObject assure le dialogue entre ces entités et les EJB via le conteneur. L'avantage de passer par le conteneur est que celui ci peut utiliser les services qu'il propose et libérer ainsi le développeur de cette charge de travail. Ceci permet au développeur de se concentrer sur les traitements métiers proposés par le bean.

Il existe de nombreux serveurs d'EJB commerciaux : BEA Weblogic, IBM Webpshere, Sun IPlanet, Macromedia JRun, Borland AppServer, etc ... Il existe aussi des serveurs d'EJB open source dont les plus avancés sont JBoss et Jonas.

51.1.1. Les différents types d'EJB

Il existe deux types d'EJB : les beans de session (session beans) et les beans entité (les entity beans). Depuis la version 2.0 des EJB, il existe un troisième type de bean : les beans orienté message (message driven beans). Ces trois types de bean possèdent des points communs notamment celui de devoir être déployés dans un conteneur d'EJB.

Les session beans peuvent être de deux types : sans état (stateless) ou avec état (stateful).

Les beans de session sans état peuvent être utilisés pour traiter les requêtes de plusieurs clients. Les beans de session avec état ne sont accessibles que lors d'un ou plusieurs échanges avec le même client. Ce type de bean peut conserver des données entre les échanges avec le client.

Les beans entité assurent la persistance des données. Il existe deux types d'entity bean :

- persistance gérée par le conteneur (CMP : Container Managed Persistence)
- persistance gérée par le bean (BMP : Bean Managed Persistence).

Avec un bean entité CMP (container-managed persistence), c'est le conteneur d'EJB qui assure la persistance des données. Un bean entité BMP (bean-managed persistence), assure lui même la persistance des données grâce à du code inclus dans le bean.

La spécification 2.0 des EJB définit un troisième type d'EJB : les beans orientés message (message-driven beans).

51.1.2. Le développement d'un EJB

Le cycle de développement d'un EJB comprend :

- la création des interfaces et des classes du bean
- le packaging du bean sous forme de fichier archive jar
- le déploiement du bean dans un serveur d'EJB
- le test du bean

La création d'un bean nécessite la création d'au minimum deux interfaces et une classe pour respecter les spécifications de Sun : la classe du bean, l'interface remote et l'interface home.

L'interface remote permet de définir l'ensemble des services fournis par le bean. Cette interface étend l'interface EJBObject. Dans la version 2.0 des EJB, l'API propose une interface supplémentaire, EJBLocalObject, pour définir les services fournis par le bean qui peuvent être appelés en local par d'autres beans. Ceci permet d'éviter de mettre en oeuvre toute une mécanique longue et couteuse en ressources pour appeler des beans s'exécutant dans le même conteneur.

L'interface home permet de définir l'ensemble des services qui vont permettre la gestion du cycle de vie du bean. Cette interface étend l'interface EJBHome.

La classe du bean contient l'implémentation des traitements du bean. Cette classe implémente les méthodes déclarées dans les interfaces home et remote. Les méthodes définissant celle de l'interface home sont obligatoirement préfixées par "ejb".

L'accès aux fonctionnalités du bean se fait obligatoirement par les méthodes définies dans les interfaces home et remote.

Il existe un certain nombre d'API qu'il n'est pas possible d'utiliser dans un EJB :

- les threads
- flux pour des entrées/sorties
- du code natif
- AWT et Swing

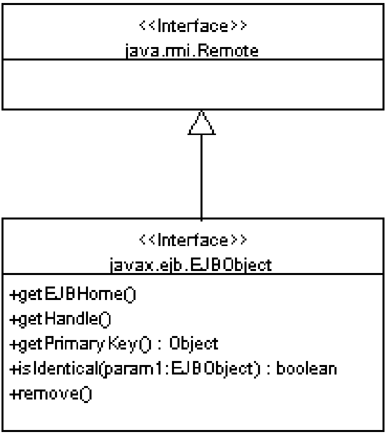
51.1.3. L'interface remote

L'interface remote permet de définir les méthodes qui contiendront les traitements proposés par le bean. Cette interface doit étendre l'interface javax.ejb.EJBObject.

Exemple :

```
package com.jmdoudoux.ejb;

import java.rmi.RemoteException;
import javax.ejb.EJBObject;

public interface MonPremierEJB extends EJBObject {
  public String message() throws RemoteException;
}
```

Toutes les méthodes définies dans cette interface doivent obligatoirement respecter les spécifications de RMI et déclarer qu'elles peuvent lever une exception de type RemoteException.



L'interface javax.ejb.EJBObject définit plusieurs méthodes qui seront donc présentes dans tous les EJB :

- EJBHome getEJBHome() throws java.rmi.RemoteException : renvoie une référence sur l'objet Home
- Handle getHandle() throws java.rmi.RemoteException : renvoie un objet permettant de sérialiser le bean
- Object getPrimaryKey() throws java.rmi.RemoteException : renvoie une référence sur l'objet qui encapsule la clé primaire d'un bean entité
- boolean isIdentical(EJBObject) throws java.rmi.RemoteException : renvoie un boolean qui précise si le bean est identique à l'instance du bean fourni en paramètre. Pour un bean session sans état, cette méthode renvoie toujours true. Pour un bean entité, la méthode renvoie true si la clé primaire des deux beans est identique.
- void remove() throws java.rmi.RemoteException, javax.ejb.RemoveException : cette méthode demande la destruction du bean. Pour un bean entité, elle provoque la suppression des données correspondantes dans la base de données.

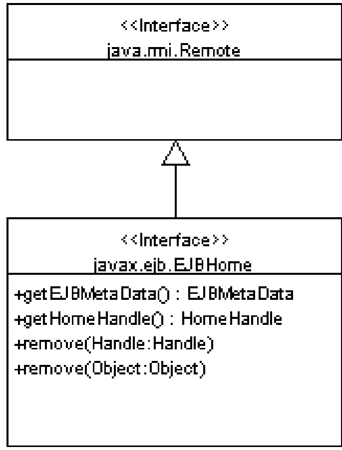
51.1.4. L'interface home

L'interface home permet de définir des méthodes qui vont gérer le cycle de vie du bean. Cette interface doit étendre l'interface EJBHome.

La création d'une instance d'un bean se fait grâce à une ou plusieurs surcharges de la méthode create(). Chacune de ces méthodes renvoie une instance d'un objet du type de l'interface remote.

Exemple :

```
package com.jmdoudoux.ejb;

import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;

public interface MonPremierEJBHome extends EJBHome {
  public MonPremierEJB create() throws CreateException, RemoteException;
}
```



L'interface javax.ejb.EJBHome définit plusieurs méthodes :

- EJBMetaData getEJBMetaData() throws java.rmi.RemoteException
- HomeHandle getHomeHandle() throws java.rmi.RemoteException : renvoie un objet qui permet de sérialiser l'objet implémentant l'interface EJBHome
- void remove(Handle) throws java.rmi.RemoteException, javax.ejb.RemoveException : supprime le bean
- void remove(Object) throws java.rmi.RemoteException, javax.ejb.RemoveException : supprime le bean entité dont l'objet encapsulant la clé primaire est fournie en paramètre

La ou les méthodes à définir dans l'interface home dépendent du type d'EJB:

Type de bean	Méthodes à définir
bean session sans état	une seule méthode create() sans paramètre
bean session avec état	une ou plusieurs méthodes create()
bean entité	aucune ou plusieurs méthodes create() et une ou plusieurs méthodes finder()

51.2. Les EJB session

Un EJB session est un EJB de service dont la durée de vie correspond à un échange avec un client. Ils contiennent les règles métiers de l'application.

Il existe deux types d'EJB session : sans état (stateless) et avec état (stateful).

Les EJB session stateful sont capables de conserver l'état du bean dans des variables d'instance durant toute la conversation avec un client. Mais ces données ne sont pas persistantes : à la fin de l'échange avec le client, l'instance de l'EJB est détruite et les données sont perdues.

Les EJB session stateless ne peuvent pas conserver de telles données entre chaque appel du client.

Il ne faut pas faire appel directement aux méthodes create() et remove() de l'EJB. C'est le conteneur d'EJB qui se charge de la gestion du cycle de vie de l'EJB et qui appelle ces méthodes. Le client décide simplement du moment de la création et de la suppression du bean en passant par le conteneur.

Une classe qui encapsule un EJB session doit implémenter l'interface javax.ejb.SessionBean. Elle ne doit pas implémenter les interfaces home et remote mais elle doit définir les méthodes déclarées dans ces deux interfaces.

La classe qui implémente le bean doit définir les méthodes définies dans l'interface remote. La classe doit aussi définir la méthode ejbCreate(), ejbRemove(), ejbActivate(), ejbPassivate et setSessionContext().

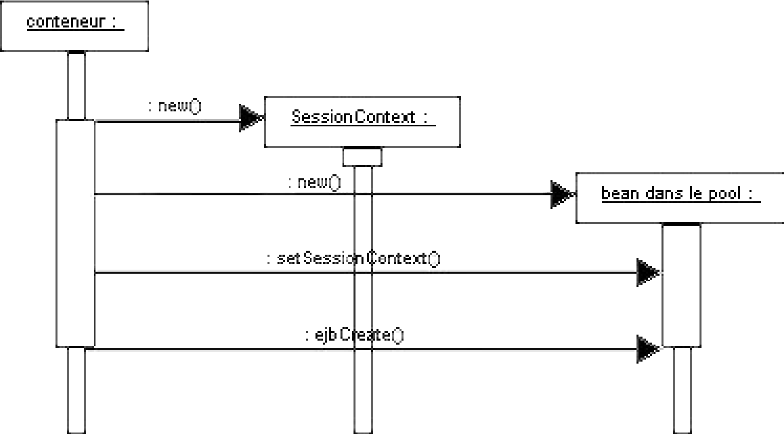
La méthode ejbRemove() est appelée par le conteneur lors de la suppression de l'instance du bean.

Pour permettre au serveur d'application d'assurer la monter en charge des différentes applications qui s'exécutent dans ces conteneurs, celui ci peut momentanément libérer de la mémoire en déchargeant un ou plusieurs beans. Cette action consiste à sérialiser le bean sur le système de fichiers et à le déssérialiser pour sa remontée en mémoire. Lors de ces deux actions, le conteneur appelle respectivement les méthodes ejbPassivate() et ejbActivate().

51.2.1. Les EJB session sans état

Ce type de bean propose des services sous la forme de méthodes. Il ne peut pas conserver de données entre deux appels de méthodes. Les données provenant du client nécessaires aux traitements d'une méthode doivent obligatoirement être fournies en paramètre de la méthode.

Les services proposés par ces beans peuvent être gérés dans un pool par le conteneur pour améliorer les performances puisqu'ils sont indépendants du client qui les utilise. Le pool contient un certain nombre d'instances du bean. Toutes ces instances étant "identiques", il suffit au conteneur d'ajouter ou de supprimer de nouvelles instances dans le pool selon les variations de la charge du serveur d'application. Il est donc inutile au serveur de sérialiser un EJB session sans état. Il suffit simplement de déclarer les méthodes ejbActivate() et ejbPassivate() sans traitements.



Le conteneur s'assure qu'un même bean ne recevra pas d'appel de méthode de la part de deux clients différents en même temps.

Exemple :

```
package com.jmdoudoux.ejb;

import java.rmi.RemoteException;
import javax.ejb.EJBException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;


public class MonPremierEJBBean implements SessionBean {

```

```
  public String message() {
    return "Bonjour";
  }

  public void ejbActivate() {
  }

  public void ejbPassivate() {
  }

  public void ejbRemove() {
  }

  public void setSessionContext(SessionContext arg0) throws EJBException, RemoteException {
  }

  public void ejbCreate() {
  }
}
```

51.2.2. Les EJB session avec état

Ce type de bean fournit aussi un ensemble de traitements via ses méthodes mais il a la possibilité de conserver des données entre les différents appels de méthodes d'un même client. Une instance particulière est donc dédiée à chaque client qui sollicite ses services et ce tout au long du dialogue entre les deux entités.

Les données conservées par le bean sont stockées dans les variables d'instances du bean. Les données sont donc conservées en mémoire. Généralement, les méthodes proposées par le bean permettent de consulter et mettre à jour ces données.

Dans un EJB session avec état il est possible de définir plusieurs méthodes permettant la création d'un tel EJB. Ces méthodes doivent obligatoirement commencer par ejbCreate.

Les méthodes ejbPassivate() et ejbActivate() doivent définir et contenir les éventuels traitements lors de leur appel par le conteneur. Celui ci appelle ces deux méthodes respectivement lors de la sérialisation du bean et sa dessérialisation. La méthode ejbActivate() doit contenir les traitements nécessaires à la restitution du bean dans un état utilisable après la dessérialisation.

Le cycle de vie d'un ejb avec état est donc identique à celui d'un bean sans état avec un état supplémentaire lorsque celui est sérialisé. La fin du bean peut être demandée par le client lorsque celui ci utilise la méthode remove(). Le conteneur invoque la méthode ejbRemove() du bean avant de supprimer sa référence.

Certaines méthodes métiers doivent permettre de modifier les données stockées dans le bean.

51.3. Les EJB entité

Ces EJB permettent de représenter et de gérer des données enregistrées dans une base de données. Ils implémentent l'interface EntityBean.

L'avantage d'utiliser un tel type d'EJB plutôt que d'utiliser JDBC ou de développer sa propre solution pour mapper les données est que certains services sont pris en charge par le conteneur.

Les beans entité assurent la persistance des données en représentant tout au partie d'une table ou d'une vue. Il existe deux types de bean entité :

- persistance gérée par le conteneur (CMP : Container Managed Persistence)
- persistance gérée par le bean (BMP : Bean Managed Persistence).

Avec un bean entité CMP (container-managed persistence), c'est le conteneur d'EJB qui assure la persistance des données grâce aux paramètres fournis dans le descripteur de déploiement du bean. Il se charge de toute la logique des traitements de synchronisation entre les données du bean et les données dans la base de données.

Un bean entité BMP (bean-managed persistence), assure lui même la persistance des données grâce à du code inclus dans les méthodes du bean.

Plusieurs clients peuvent accéder simultanément à un même EJB entity. La gestion des transactions et des accès concurrents est assurée par le conteneur.

51.4. Les outils pour développer et mettre oeuvre des EJB

La mise en oeuvre des EJB requiert un conteneur d'EJB généralement inclus dans un serveur d'applications et un IDE pour être productif.

51.4.1. Les outils de développement

Plusieurs EDI (Environnement de Développement Intégré) open source permettent de développer et de tester des EJB notamment Eclipse et Netbeans. Netbeans est d'ailleurs celui qui propose le plus rapidement une implémentation pour mettre en oeuvre la dernière version des spécifications relatives aux EJB.

51.4.2. Les conteneurs d'EJB

Il existe plusieurs conteneurs d'EJB commerciaux mais aussi d'excellent conteneur d'EJB open source notamment Glassfish, JBoss ou Jonas.

51.4.2.1. JBoss

JBoss est un serveur d'applications Java EE open source écrit en Java.

Il peut être téléchargé sur www.jboss.org.

Pour l'installer, il suffit de décompresser l'archive et de copier son contenu dans un répertoire , par exemple : c\jboss

Pour lancer le serveur, il suffit d'exécuter la commande :

java -jar run.jar

Les EJB à déployer doivent être mis dans le répertoire deploy. Si le répertoire existe au lancement du serveur, les EJB seront automatiquement déployés dès qu'ils seront insérés dans ce répertoire.

51.5. Le déploiement des EJB

Un EJB doit être déployé sous forme d'une archive jar qui doit contenir un fichier qui est le descripteur de déploiement et toutes les classes qui composent chaque EJB (interfaces home et remote, les classes qui implémentent ces interfaces et toutes les autres classes nécessaires aux EJB).

Une archive ne doit contenir qu'un seul descripteur de déploiement pour tous les EJB de l'archive. Ce fichier au format XML doit obligatoirement être nommé ejb-jar.xml.

L'archive doit contenir un répertoire META-INF (attention au respect de la casse) qui contiendra lui même le descripteur de déploiement.

Le reste de l'archive doit contenir les fichiers .class avec toute l'arborescence des répertoires des packages.

Le jar des EJB peut être inclus dans un fichier de type EAR.

51.5.1. Le descripteur de déploiement

Le descripteur de déploiement est un fichier au format XML qui permet de fournir au conteneur des informations sur les beans à déployer. Le contenu de ce fichier dépend du type de beans à déployer.

51.5.2. Le mise en package des beans

Une fois toutes les classes et le fichier de déploiement écrit, il faut les rassembler dans une archive .jar afin de pouvoir les déployer dans le conteneur.

51.6. L'appel d'un EJB par un client

Un client peut être une entité de toute forme : application avec ou sans interface graphique, un bean, une servlet ou une JSP ou un autre EJB.

Un EJB étant un objet distribué, son appel utilise RMI.

Le stub est une représentation locale de l'objet distant. Il implémente l'interface remote mais contient une connexion réseau pour accéder au skeleton de l'objet distant.

Le mode d'appel d'un EJB suit toujours la même logique :

- obtenir une référence qui implémente l'interface home de l'EJB grâce à JNDI
- créer une instance qui implémente l'interface remote en utilisant la référence précédemment acquise
- appelle de la ou des méthodes de l'EJB

51.6.1. Un exemple d'appel d'un EJB session

L'appel d'un EJB session avec ou sans état suit la même logique.

Il faut tout d'abord utiliser un objet du type InitialContext pour pouvoir interroger JNDI. Cet objet nécessite qu'on lui fournisse des informations dont le nom de la classe à utiliser comme fabrique et l'url du serveur JNDI.

Cet objet permet d'obtenir une référence sur le bean enregistré dans JNDI. A partir de cette référence, il est possible de créer un objet qui implémente l'interface home. Un appel à la méthode create() sur cet objet permet de créer un objet du type de l'EJB. L'appel des méthodes de cet objet entraine l'appel des méthodes de l'objet EJB qui s'exécute dans le conteur.

Exemple :

```
package testEJBClient;

import java.util.*;
import javax.naming.*;

public class EJBClient {
```

```
  public static void main(String[] args) {
    Properties ppt = null;
    Context ctx = null;
    Object ref = null;
    MonPremierBeanHome home = null;
    MonPremierBean bean = null;

    try {
      ppt = new Properties();
      ppt.put(Context.INITIAL_CONTEXT_FACTORY, "org.jnp.interfaces.NamingContextFactory");
      ppt.put(Context.PROVIDER_URL, "localhost:1099");
      ctx = new InitialContext(ppt);
      ref = ctx.lookup("MonPremierBean");
      home = (MonPremierBeanHome) javax.rmi.PortableRemoteObject.narrow(ref,
        MonPremierBeanHome.class);
      bean = home.create();
      System.out.println("message = " + bean.message());
      bean.remove();
    }
    catch (Exception e) {
       e.printStackTrace();
    }
  }
}
```

51.7. Les EJB orientés messages

Ces EJB sont différents des deux types d'EJB car ils répondent à des invocations de façon asynchrone. Ils permettent de réagir à l'arrivée de messages fournis par un M.O.M. (Middleware Oriented Messages).

52. Les EJB 3

Les EJB (Entreprise Java Bean) sont un des éléments très important de la plate-forme Java EE pour le développement d'applications distribuées.

La plate-forme Java EE propose de mettre en oeuvre les couches métiers et persistance avec les EJB. Particulièrement intéressants dans des environnements fortement distribués, leur mise en oeuvre est assez lourde jusqu'à la version 3 sans l'utilisation d'outils tels que certains IDE ou XDoclet.

La version 3 des EJB vise donc à simplifier le développement et la mise en oeuvre des EJB qui sont fréquemment jugés trop complexes et trop lourds à mettre en oeuvre.

Cette nouvelle version majeure des EJB propose une simplification de leur développement tout en conservant une compatibilité avec la précédente version des EJB. Elle apporte de très nombreuses fonctionnalités dans le but de simplifier la mise en oeuvre des EJB.

Cette simplification est rendue possible notamment par :

- l'utilisation des annotations
- la mise en oeuvre de valeurs par défaut qui répond à la plupart des besoins (configuration par exception)
- le descripteur de déploiement est facultatif
- l'utilisation de POJO et de JPA pour les beans de type entity
- l'injection de dépendances côté serveur mais aussi côté client (l'interface Home qui gérait le cycle de vie est abandonnée) qui remplace l'utilisation directe de JNDI
- ...

Tous ces éléments délèguent une partie du travail du développeur au conteneur d'EJB.

52.1. L'historique des EJB

EJB 1.1 publié en décembre 1999, intégré dans J2EE 1.2 :

- Session beans (stateless/stateful)
- Entity Beans (CMP / BMP)
- Interface Remote uniquement

EJB 2.0 publié en septembre 2001, intégré à J2EE 1.3 :

- Message-Driven Beans
- Entity 2.x reposant sur EJB QL
- Interface Local pour améliorer les performances des appels dans la même JVM

EJB 2.1 publié en novembre 2003, intégré à J2EE 1.4 :

- EJB Timer Service
- EJB Web Service Endpoints via JAX-RPC
- Amélioration du langage EJB QL

EJB 3.0, intégré à Java EE 5 :

- utilisation de POJO et POJI, plus d'interface Home
- utilisation des annotations, le descripteur de déploiement est optionnel
- utilisation de JPA pour les beans de type entity

EJB 3.1, intégré à Java EE 6

52.2. Les nouveaux concepts et fonctionnalités utilisés

Dans les versions antérieures à la version 3.0 des EJB, le développeur était contraint de créer de nombreuses entités pour respecter l'API EJB (par exemple, l'implémentation d'interfaces engendrant la création de plusieurs méthodes ou le descripteur de déploiement), ce qui rendait leur écriture relativement lourde même avec l'assistance de certains IDE ou outils (XDoclet notamment). Dans la version 3.0, ceci est remplacé par l'utilisation d'annotations.

La mise en oeuvre de l'interface EJBHome n'est plus requise : un EJB de type session est maintenant une simple classe, qui peut implémenter une interface métier.

La seule annotation obligatoire dans un EJB est celle qui précise le type d'EJB (@javax.ejb.Stateless, @javax.ejb.Stateful ou @javax.ejb.MessageDriven).

Les annotations possèdent des valeurs par défaut qui répondent à une majorité de cas typique d'utilisation. L'utilisation de ces annotations n'est alors requise que si les valeurs par défaut ne répondent pas au besoin. Ceci permet de réduire la quantité de code à écrire.

L'utilisation des annotations et de valeurs par défaut pour la plupart de ces dernières rend optionnelle la nécessité de créer un descripteur de déploiement sauf pour des besoins très particuliers.

Le conteneur obtient des informations sur la façon de mettre en oeuvre un EJB par trois moyens :

- Des valeurs par défaut pour la plupart des annotations ce qui évite d'avoir à les utiliser explicitement dans le code
- Les annotations utilisées dans le code
- Le descripteur de déploiement

L'ordre d'utilisation par le conteneur est : le descripteur de déploiement, les annotations, les valeurs par défaut.

L'utilisation des annotations est plus simple à mettre en oeuvre mais le descripteur de déploiement permet de centraliser les informations.

La nouvelle API Java Persistence remplace la persistance assurée par le conteneur : cette API assure la persistance des données grâce à un mapping O/R reposant sur des POJO.

Le conteneur a la possibilité d'injecter des dépendances d'objets gérés par le conteneur.

Les intercepteurs permettent d'offrir certaines fonctionnalités proches de certaines proposées par l'AOP : ceci permet de définir des traitements lors de l'invocation de méthodes des EJB ou d'invoquer certaines méthodes liées au cycle de vie de l'EJB.

52.2.1. L'utilisation de POJO et POJI

Les classes et les interfaces des EJB 3.0 sont de simples POJO ou POJI : ceci simplifie le développement des EJB. Par exemple, l'interface Home n'est plus à déclarer.

Il est toujours possible d'implémenter les interfaces SessionBean, EntityBean et MessageDrivenBean mais le plus simple est d'utiliser les annotations définies : @Stateless, @Stateful, @Entity ou @MessageDriven

Exemple :

```
@Stateless
public class HelloWorldBean {
  public String saluer(String nom)
  {
    return "Bonjour "+nom;
  }
}
```

Il est possible de définir une interface métier pour l'EJB ou de laisser générer cette interface lors du déploiement.

Dans le premier cas, il n'est plus nécessaire qu'elle implémente l'interface EJBObject ou EJBLocalObject mais simplement d'utiliser les annotations définies : @Remote ou @Local.

Exemple :

```
@Remote
@Stateless
public class HelloWorldBean {
  public String saluer(String nom)
  {
    return "Bonjour "+nom;
  }
}
```

Dans le second cas, ces annotations doivent être utilisées dans la classe d'implémentation pour permettre de déterminer l'interface générée.

Il est possible de définir une interface locale et/ou distante pour un même EJB.

Il n'est pas recommandé de laisser les interfaces être générées par le conteneur pour plusieurs raisons :

- les interfaces générées exposent par défaut toutes les méthodes de l'EJB
- l'interface est utilisée par le client pour invoquer l'EJB
- le nom des interfaces générées utilise le nom de l'implémentation de l'EJB

52.2.2. L'utilisation des annotations

La spécification 3.0 des EJB fait un usage intensif des annotations. Celles-ci sont issues de la JSR 175 et intégrées dans Java SE 5.0 qui constitue la base de Java EE 5.

Les annotations sont des attributs ou méta-données à l'image de celles proposées par XDoclet.

Avec les EJB 3.0, les annotations sont utilisées pour générer des entités et remplacer tout ou partie du descripteur de déploiement.

De nombreuses annotations permettent de simplifier le développement des EJB.

La nature de l'EJB est précisée par une des annotations @Stateless, @Stateful, @Entity et @MessageDriven selon le type d'EJB à définir.

Le type d'accès est précisé par deux annotations

- @Remote : permet un accès à l'EJB depuis un client hors de la JVM
- @Local : permet un accès à l'EJB depuis un client dans la même JVM que celle de l'EJB

Par défaut, l'interface d'appel est locale si aucune annotation n'est indiquée.

Dans le cas d'un accès distant, il est inutile que chaque méthode précise qu'elle peut lever une exception de type RemoteException mais elles peuvent déclarer la levée d'exceptions métier.

Jusqu'à la version 2.1 des EJB, il était obligatoire d'implémenter plusieurs méthodes relatives à la gestion du cycle de vie de l'EJB notamment ejbActivate, ejbLoad, ejbPassivate, ejbRemove, ... pour chaque EJB même si ces méthodes ne contenaient aucun traitement.

Avec les EJB 3.0, l'implémentation de ces méthodes est remplacée par l'utilisation facultative selon les besoins d'annotations sur les méthodes concernées. La signature de ces méthodes doit être de la forme public void nomMethode()

Par exemple, pour que le conteneur exécute automatiquement une méthode avant de retirer l'instance du bean, il faut annoter la méthode avec l'annotation @Remove.

Plusieurs annotations permettent ainsi de définir des méthodes qui interviendront dans le cycle de vie de l'EJB.

Annotation	Rôle
@PostConstruct	est invoquée après que l'instance soit créée et que les dépendances soient injectées
@PostActivate	est invoquée après que l'instance de l'EJB ne soit déssérialisée du disque. C'est l'équivalent de la méthode ejbActivate() des EJB 2.x
@Remove	est invoquée avant que l'EJB ne soit retiré du conteneur
@PreDestroy	est invoquée avant que l'instance de l'EJB ne soit supprimée
@PrePassivate	est invoquée avant de l'instance de l'EJB ne soit sérialisée sur disque. C'est l'équivalent de la méthode ejbPassivate() des EJB 2.x

L'utilisation facultative de ces annotations remplace la définition obligatoire des méthodes de gestion du cycle de vie utilisées jusqu'à la version 2.1 des EJB.

Le descripteur de déploiement n'est plus obligatoire puisqu'il peut être remplacé par l'utilisation d'annotations dédiées directement dans les classes des EJB.

Chaque attribut de déploiement possède une valeur par défaut qu'il ne faut définir que si cette valeur ne répond pas au besoin.

Plusieurs annotations sont définies par les spécifications des EJB pour permettre de définir le type de bean, le type de l'interface, des références vers des ressources qui seront injectées, la gestion des transactions, la gestion de la sécurité, ...

Chaque vendeur peut définir en plus ces propres annotations dans l'implémentation de son serveur d'application. Leur utilisation n'est cependant pas recommandée car ils rendent l'application dépendante du serveur d'applications utilisé.

L'utilisation des annotations va simplifier le développement des EJB mais la gestion de la configuration pourra devenir plus complexe puisqu'elle n'est plus centralisée.

52.2.3. L'injection de dépendances

L'EJB déclare les ressources dont il a besoin à l'aide d'annotations. Le conteneur va injecter ces ressources lors qu'il va instancier l'EJB donc avant l'appel aux méthodes liées au cycle de vie du bean ou aux méthodes métiers. Ceci impose que l'injection de ressources ne peut se faire que sur des objets qui sont gérés par le conteneur.

Ces ressources peuvent être de diverses natures : référence vers un autre EJB, contexte de sécurité, contexte de persistance, contexte de transaction, ...

Plusieurs annotations sont définies pour mettre en oeuvre l'injection de dépendances :

- L'annotation @EJB permet d'injecter une ressource de type EJB.

* L'annotation @Resource permet d'injecter une ressource qui est stockée via JNDI (EntityManager, UserTransaction, SessionContext, ...)
* ...

L'utilisation de l'injection de dépendances remplace l'utilisation implicite de JNDI.

L'injection peut aussi être définie dans le descripteur de déploiement.

52.2.4. La configuration par défaut

La suite de cette section sera développée dans une version future de ce document

52.2.5. Les intercepteurs

Les intercepteurs sont une fonctionnalité avancée similaire à celle proposée par l'AOP : elle permet d'intercepter l'invocation de méthodes pour exécuter des traitements.

Ils sont définis grâce à des annotations dédiées notamment @Interceptors et @AroundInvoke ou dans le descripteur de déploiement.

Leur utilisation peut être variée : traces, logs, gestion de la sécurité, ...

52.3. EJB 2.x vs EJB 3.0

Le développement d'EJB n'a jamais été facile et est même devenu plus complexe au fur et à mesure des nouvelles spécifications.

Avant la version 3.0 des EJB, les EJB sont relativement complexes et lourds à mettre en oeuvre :

* création de plusieurs interfaces et classes (deux interfaces et une classe au minimum)
* implémentation de méthodes callback généralement inutiles
* l'interface de l'EJB doit hériter de EJBObject ou de EJBLocalObject
* chaque méthode de l'EJB doit déclarer pouvoir lever l'exception RemoteException
* le descripteur de déploiement des EJB est complexe
* les EJB Entité de type CMP possèdent plusieurs limitations : complexe à développer et à maintenir, de nombreux problèmes de performance, le langage EJBQL est limité
* le support de la POO pour les EJB est très limité vis-à-vis de l'héritage
* les EJB doivent être testés dans un conteneur ce qui les rend difficile à déboguer
* l'appel d'un EJB par un client nécessite obligatoirement une utilisation de JNDI

La version 3.0 des spécifications des EJB apporte une solution de simplification à tous les points précédemment cités.

* Il n'est plus nécessaire de déclarer d'interfaces (pour des raisons de bonne pratique, la déclaration d'une interface métier contenant les méthodes proposées est cependant fortement recommandée)
* Le descripteur de déploiement est optionnel sauf dans des cas particuliers
* L'utilisation de POJO et POJI
* L'injection de dépendances rend très facile l'obtention d'une instance d'une ressource gérée par le conteneur

* ...

Les principales différences entre les EJB 2.x et EJB 3.0 sont donc :

* les descripteurs de déploiement ne sont plus obligatoires grâce à l'utilisation d'annotations et de valeurs par défaut
* Les EJB sont de simples POJO annotés : ils n'ont plus besoin d'implémenter une interface de l'API EJB. De fait, il n'est plus nécessaire de définir des méthodes liées au cycle de vie de l'EJB. Si ces méthodes sont nécessaires, il suffit d'utiliser des annotations dédiées sur une méthode.
* Le type de l'interface de l'EJB est précisé avec l'annotation @Local ou @Remote
* L'interface métier est une simple POJI
* Les EJB de type Entity CMP et BMP sont remplacés par l'utilisation du modèle de persistance reposant sur l'API JPA

52.4. Les conventions de nommage

Il n'existe pas de règles imposées mais il est important de définir des conventions de nommage pour les différentes entités qui sont utilisées lors de la mise en oeuvre des EJB.

Exemple :

Nom du bean : CalculEJB

Nom de la classe métier : CalculBean

Interface locale : CalculLocal

Interface distante : CalculRemote

52.5. Les EJB de type Session

Les EJB session sont généralement utilisés comme façade pour proposer des fonctionnalités qui peuvent faire appel à d'autres composants ou entités tels que des EJB session, des EJB Entity, des POJO, ...

La version 3.0 des EJB rend inutile l'implémentation d'une interface spécifique à l'API EJB. Mais même si cela n'est pas obligatoire, il est fortement recommandé (dans la mesure du possible) de définir une interface dédiée à l'EJB qui va notamment préciser son mode d'accès et les méthodes utilisables.

Cette interface est alors une simple POJI.

52.5.1. L'interface distante et/ou locale

Un EJB peut être invoqué :

* en local : le client appelant est exécuté dans la même JVM que celle de l'EJB. Ce type d'appel est le plus performant puisqu'il ne nécessite pas d'échanges réseaux et donc pas de mécanisme pour gérer ces échanges
* à distance : le client appelant est exécuté dans une autre JVM que celle de l'EJB

L'interface distante définit les méthodes qui peuvent être appelées par un client en dehors de la JVM du conteneur. L'interface ou le bean doit être marquée avec l'annotation @Remote implémentée dans la classe javax.ejb.Remote

L'interface locale définit les méthodes qui peuvent être appelées par un autre EJB s'exécutant dans la même JVM que le conteneur. Les performances sont ainsi accrues car les mécanismes de protocoles d'appels distants ne sont pas utilisés (sérialisation/désérialisation, RMI, ...).

L'utilisation de l'interface Local pour des appels à l'EJB dans un même JVM est fortement recommandé par cela améliore les performances de façon dramatique. L'interface Remote met en oeuvre des mécanismes de communication utilisant la sérialisation, ce qui dégrade les performances notamment de façon inutile si l'appel à l'EJB se fait dans une même JVM.

Un client ne dialogue jamais en direct avec une instance de l'EJB : le client utilise toujours l'interface pour accéder au bean grâce à un proxy généré par le conteneur. Même un client local utilise un proxy particulier dépourvu des accès réseau. Ce proxy permet au conteneur d'assurer certaines fonctionnalités comme la sécurité et les transactions.

52.5.2. Les beans de type Stateless

Les beans de type stateless sont les plus simples et les plus véloces car le conteneur gère un pool d'instances qui sont utilisées au besoin, ce qui évite des opérations d'instanciation et de destruction à chaque utilisation. Ceci permet une meilleure montée en charge de l'application.

L'annotation @javax.ejb.Stateless permet de préciser qu'un EJB session est de type stateless. Elle s'utilise sur une classe qui encapsule un EJB et possède plusieurs attributs :

Attribut	Rôle
String name	Nom de l'EJB (optionnel)
String mappedName	Nom sous lequel l'EJB sera mappé. La valeur par défaut est le nom non qualifié de la classe (optionnel)
String description	Description de l'EJB (optionnel)

Il faut définir l'interface de l'EJB avec l'annotation précisant le mode d'accès

L'annotation @javax.ejb.Remote permet de préciser que l'EJB pourra être accédé par des clients distants. Elle s'utilise sur une classe qui encapsule un EJB ou l'interface qui décrit les fonctionnalités de l'EJB utilisables à distance. Cette annotation ne peut être utilisée que pour des EJB sessions.

Elle possède un seul attribut :

Attribut	Rôle
Class[] value	Préciser la liste des interfaces distantes de l'EJB. Son utilisation est obligatoire si la classe de l'EJB implémente plusieurs interfaces différentes java.io.Serializable, java.io.Externalizable, ou une des interfaces du package javax.ejb (optionnel)

Exemple :

```
import javax.ejb.Remote;

@Remote
public interface CalculRemote {
  public long additionner(int valeur1, int valeur2);
}
```

Cette interface est marquée avec l'annotation @Remote pour permettre un appel distant et définit la méthode additionner.

Remarque : l'utilisation de l'annotation rend inutile l'utilisation de la clause throws RemoteException des versions antérieures des EJB.

L'annotation @javax.ejb.Local permet de préciser que l'EJB pourra être accédé par des clients locaux de la JVM. Elle s'utilise sur une classe qui encapsule un EJB ou l'interface qui décrit les fonctionnalités de l'EJB utilisables en local dans la JVM. Cette annotation ne peut être utilisée que pour des EJB sessions.

Elle possède un attribut :

Attribut	Rôle
Class[] value	Préciser la liste des interfaces distantes de l'EJB. Son utilisation est obligatoire si la classe de l'EJB implémente plusieurs interfaces différentes java.io.Serializable, java.io.Externalizable, ou une des interfaces du package javax.ejb (optionnel)

Exemple :

```
import javax.ejb.Local;

@Local
public interface CalculLocal {
  public long additionner(int valeur1, int valeur2);
}
```

Cette interface est marquée avec l'annotation @Local pour permettre un appel local et définit la méthode additionner.

Il faut ensuite définir la classe de l'EJB qui va contenir les traitements métier.

Exemple :

```
import javax.ejb.*;

@Stateless
public class CalculBean implements CalculRemote, CalculLocal {
  public long additionner(int valeur1, int valeur2) {
    return valeur1 + valeur2;
  }
}
```

Cette classe est marquée avec l'annotation @Stateless et implémente les interfaces distante et locale précédemment définies.

Il est préférable lorsque cela est possible d'utiliser l'interface Local car elle est beaucoup plus performante. L'interface Remote est à utiliser lorsque le client n'est pas dans la même JVM.

Les annotations @Local et @Remote peuvent être utilisées directement sur l'EJB mais il est préférable de définir une interface par mode d'accès et d'utiliser l'annotation adéquate sur chacune des interfaces.

La classe de l'EJB ne doit plus implémenter l'interface javax.ejb.SessionBean qui était obligatoire avec les EJB 2.x. Maintenant, les EJB session de type stateless peuvent utiliser les callbacks d'évènements marqués avec les annotations suivantes :

- @PostConstruct
- @PreDestroy

52.5.3. Les beans de type stateful

Les beans de type stateful sont capables de conserver leur état durant toute leur utilisation par le client. Cet état n'est cependant pas persistant : les données sont perdues à la fin de son utilisation ou à l'arrêt du serveur. Un exemple type d'utilisation de ce type de bean est l'implémentation d'un caddie pour un site de vente en ligne.

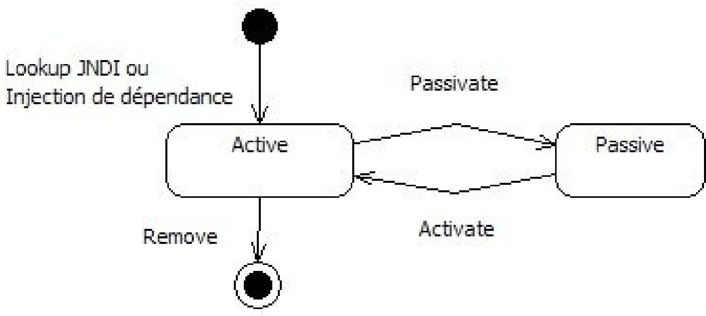
L'annotation @javax.ejb.Stateful permet de préciser qu'un EJB session est de type stateful. Elle s'utilise sur une classe qui encapsule un EJB.

Elle possède plusieurs attributs :

Attribut	Rôle

String name	Nom de l'EJB (optionnel)
String mappedName	Nom sous lequel l'EJB sera mappé. La valeur par défaut est le nom non qualifié de la classe (optionnel)
String description	Description de l'EJB (optionnel)

Le conteneur EJB a la possibilité de serialiser/dessérialiser des EJB de type Stateful notamment dans le cas ou la JVM du conteneur commence à manquer de mémoire. Dans ce cas, le conteneur peut serialiser des EJB (passivate) qui ne sont pas en cours d'utilisation sur disque. Dès qu'un de ces EJB sera sollicité, le conteneur va le desserialiser (activate) du disque pour le remettre en mémoire et pouvoir l'utiliser.



Ils n'ont plus à implémenter l'interface javax.ejb.SessionBean comme c'était le cas dans les versions antérieures aux EJB 3.0. Maintenant, les EJB session de type stateful peuvent utiliser les callbacks d'évènements marqués avec les annotations suivantes :

- @PostConstruct
- @PostActivate
- @PreDestroy
- @PrePassivate
- @Remove

52.5.4. L'invocation d'un EJB Session via un service web

Le support des services web dans les EJB 3.0 repose essentiellement sur JAX-WS 2.0 et SAAJ qu'il faut privilégier au détriment de JAX-RPC qui est toujours supporté.

La suite de cette section sera développée dans une version future de ce document

52.5.5. L'utilisation des exceptions

Les exceptions personnalisées qui sont utilisées dans les interfaces métier des EJB doivent être annotées avec l'annotation @javax.ejb.ApplicationException. Elle s'utilise donc sur une classe qui encapsule une exception métier.

Une exception annotée avec @ApplicationException sera directement envoyée au client par le conteneur.

Elle possède un seul attribut :

Attribut	Rôle
boolean rollback	Préciser si le conteneur doit effectuer un rollback si cette exception est levée. La valeur par défaut est false (optionnel)

L'attribut rollback de l'annotation @ApplicationException de type booléen permet de préciser si la levée de l'exception va déclencher ou non un rollback de la transaction en cours. La valeur par défaut est false, signifiant qu'il n'y aura pas de rollback.

Exemple :

```
package com.jmd.test.domaine.ejb;

import javax.ejb.ApplicationException;

@ApplicationException
public class ErreurMetierException extends Exception {

    public ErreurMetierException() {
    }


    public ErreurMetierException(String msg) {
        super(msg);
    }
}
```

L'annotation @ApplicationException peut être utilisée avec des exceptions de type checked et unchecked.

52.6. Les EJB de type Entity

Dans les versions antérieures des EJB, les EJB de type Entity avaient la charge de la persistance des données. Les EJB de type Entity CMP (Container Managed Persistence) doivent simplement requérir un fichier de description.

Les EJB 3.0 proposent d'utiliser l'API Java Persistence pour assurer la persistance des données dans les EJB : ils utilisent un modèle de persistance léger standard en remplacement des entity beans de type CMP.

JPA repose sur des beans entity qui sont de simples POJO enrichis d'annotations qui permettent de mettre en oeuvre les concepts de POO tels que l'héritage ou le polymorphisme.

Jusqu'à la version 3.0 des EJB, les Entity beans sont des composants qui dépendent pleinement du conteneur d'EJB du serveur d'applications dans lequel ils s'exécutent. L'utilisation de POJO avec l'API Java Persistence permet de rendre les beans entity indépendants du conteneur. Ceci possède plusieurs avantages dont celui de pouvoir facilement tester les beans puisqu'ils ne requièrent plus de conteneur pour leur exécution.

Avec la version 3.0 des EJB, les beans entity sont donc des POJO qui n'ont donc pas besoin d'implémenter une interface spécifique aux EJB et qui doivent posséder un constructeur sans argument et implémenter l'interface Serializable.

Les attributs persistants sont déclarés via des annotations soit au niveau de l'attribut soit au niveau de son getter/setter. De ce fait, ils peuvent être utilisés directement comme objets du domaine ; il n'y a plus l'obligation de définir un DTO.

52.6.1. La création d'un bean Entity

Les beans de type Entity sont dans la version 3.0 des spécifications de simple POJO utilisant les annotations de l'API Java Persistence (JPA) pour définir le mapping.

Les informations de mapping entre une table et un objet peuvent être définies grâce aux annotations mais aussi via un fichier de mapping qui permet d'externaliser ces informations du POJO. Il est possible de mixer les deux (annotations et fichiers de mapping) mais les données incluses dans le fichier sont prioritaires par rapport aux annotations.

Le bean entity doit être annoté avec l'annotation @Entity implémentée dans la classe javax.persistence.Entity.

L'annotation @Table implémentée dans la classe javax.persistence.Table permet de préciser le nom de la table vers lequel le bean sera mappé. L'utilisation de cette annotation est facultative si le nom de la table correspond au nom de la classe.

Pour mapper un champ de la table avec une propriété du bean, il faut utiliser l'annotation @Column implémentée dans la classe javax.persistence.Column sur le getter de la propriété. L'utilisation de cette annotation est facultative si le nom du champ correspond au nom de la propriété.

Le champ correspondant à la clé primaire de la table doit être annoté avec l'annotation @Id implémenté dans la classe javax.persistence.Id. L'utilisation de cette annotation est obligatoire car un identifiant unique est obligatoire pour chaque occurrence et l'API n'a aucun moyen de déterminer le champ qui encapsule cette information.

Il peut être pratique pour un bean de type entity d'implémenter l'interface Serializable : le bean pourra être utilisé dans les paramètres et la valeur de retour des méthodes métiers d'un EJB. Le bean peut ainsi être utilisé pour la persistance et le transfert de données.

Exemple :

```
package com.jmd.test.domaine.entity;

import java.io.Serializable;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;
import javax.persistence.Table;

@Entity
@Table(name = "PERSONNE")
@NamedQueries({@NamedQuery(name = "Personne.findById",
  query = "SELECT p FROM Personne p WHERE p.id = :id"),
               @NamedQuery(name = "Personne.findByNom",
  query = "SELECT p FROM Personne p WHERE p.nom = :nom"),
               @NamedQuery(name = "Personne.findByPrenom",
  query = "SELECT p FROM Personne p WHERE p.prenom = :prenom")})
public class Personne implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @Column(name = "ID", nullable = false)
    private Integer id;
    @Column(name = "NOM")
    private String nom;
    @Column(name = "PRENOM")
    private String prenom;

    public Personne() {
    }

    public Personne(Integer id) {
        this.id = id;
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getNom() {
        return nom;
```

```
    }

    public void setNom(String nom) {
        this.nom = nom;
    }

    public String getPrenom() {
        return prenom;
    }

    public void setPrenom(String prenom) {
        this.prenom = prenom;
    }

    @Override
    public String toString() {
        return "com.jmd.test.domaine.entity.Personne[id=" + id + "]";
    }

}
```

Remarque : il est préférable de définir tous les beans de type entity dans un package dédié.

La mise en oeuvre précise de l'API JPA est proposée dans le chapitre qui lui est consacrée.

52.6.2. La persistance des entités

La version 3.0 propose une refonte complète des EJB entités afin de simplifier leur développement. Cette simplification est assurée en grande partie par la mise en oeuvre de JPA qui permet :

- la standardisation du mapping O/R
- l'utilisation de POJO annotés avec support de l'héritage et du polymorphisme
- la possibilité d'utiliser les EJB entités en dehors du conteneur d'EJB qui permet notamment la mise en oeuvre de tests unitaires automatisés.

La persistance d'objets avec JPA repose sur plusieurs fonctionnalités :

- un ensemble d'entités annotées qui représente le modèle objet du domaine
- une API contenue dans le package javax.persistence
- un cycle de vie pour les entités

La classe EntityManager est responsable de la gestion des opérations sur une entité notamment grâce à plusieurs méthodes :

- persist()
- remove()
- merge()
- flush()
- find()
- refresh()
- ...

La suite de cette section sera développée dans une version future de ce document

52.6.3. La création d'un EJB Session pour manipuler le bean Entity

Le bean entity n'est utilisé que pour le mapping. Pour réaliser des opérations avec le bean entity, il faut développer un EJB Session qui va encapsuler la logique des traitements à réaliser avec le bean entity.

Il faut définir l'interface local et/ou remote des méthodes métier de l'EJB.

Il faut ensuite définir l'EJB qui va utiliser l'API Java Persistence et le bean entity.

L'injection de dépendances est utilisée pour obtenir une instance de l'EntityManager par le conteneur.

Exemple :

```
        @PersistenceContext
        private EntityManager em;
```

L'annotation @PersistenceContext demande au conteneur d'injecter une instance de la classe EntityManager.

Le conteneur retrouve l'EntityManager grâce au nom de l'unité de persistnce fournie comme valeur à la propriété unitName de l'annotation si plusieurs unités de persistance sont définies.

L'instance de type EntityManager peut être utilisée dans les méthodes métier pour réaliser des traitements sur le bean entity.

Exemple :

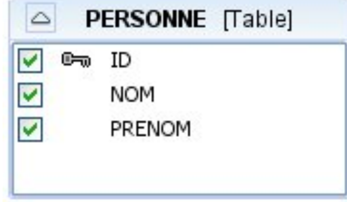
```
...
    public void create(Personne personne) {
        em.persist(personne);
    }

    public void edit(Personne personne) {
        em.merge(personne);
    }

    public void remove(Personne personne) {
        em.remove(em.merge(personne));
    }

    public Personne find(Object id) {
        return em.find(com.jmd.test.domaine.entity.Personne.class, id);
    }

...
```

52.7. Un exemple simple complet

L'exemple de cette section va développer un EJB métier permettant des opérations de type CRUD sur une table nommée personne et permettre l'appel de cet EJB via un service web.

La table personne contient trois champs :

- id : identifiant unique de la personne
- nom : nom de la personne
- prenom : prénom de la personne



Cette table est stockée dans une base de données de type JavaDB.

Une connexion vers la base de données est définie dans l'annuaire sous le nom MaTestDb

Remarque : les sources de cet exemple sont générées par l'IDE Netbeans.

52.7.1. La création de l'entité

La classe Personne encapsule une entité sur la table personne.

Exemple :

```
package com.jmd.test.domaine.entity;

import java.io.Serializable;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;
import javax.persistence.Table;

@Entity
@Table(name = "PERSONNE")
@NamedQueries({@NamedQuery(name = "Personne.findById",
  query = "SELECT p FROM Personne p WHERE p.id = :id"),
               @NamedQuery(name = "Personne.findByNom",
  query = "SELECT p FROM Personne p WHERE p.nom = :nom"),
               @NamedQuery(name = "Personne.findByPrenom",
  query = "SELECT p FROM Personne p WHERE p.prenom = :prenom")})
public class Personne implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @Column(name = "ID", nullable = false)
    private Integer id;
    @Column(name = "NOM")
    private String nom;
    @Column(name = "PRENOM")
    private String prenom;

    public Personne() {
    }

    public Personne(Integer id) {
        this.id = id;
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getNom() {
        return nom;
    }

    public void setNom(String nom) {
```

```
        this.nom = nom;
    }

    public String getPrenom() {
        return prenom;
    }

    public void setPrenom(String prenom) {
        this.prenom = prenom;
    }

    @Override
    public String toString() {
        return "com.jmd.test.domaine.entity.Personne[id=" + id + "]";
    }

}
```

52.7.2. La création de la façade

L'interface métier locale est définie dans l'interface PersonneFacadeLocal

Exemple :

```
package com.jmd.test.domaine.ejb;

import com.jmd.test.domaine.entity.Personne;
import java.util.List;
import javax.ejb.Local;

@Local
public interface PersonneFacadeLocal {

    void create(Personne personne);

    void edit(Personne personne);

    void remove(Personne personne);

    Personne find(Object id);

    List<Personne> findAll();

}
```

L'interface métier distante est définie dans l'interface PersonneFacadeRemote

Exemple :

```
package com.jmd.test.domaine.ejb;

import com.jmd.test.domaine.entity.Personne;
import java.util.List;
import javax.ejb.Remote;

@Remote
public interface PersonneFacadeRemote {

    void create(Personne personne);

    void edit(Personne personne);

    void remove(Personne personne);

    Personne find(Object id);

```

```
    List<Personne> findAll();

}
```

La façade est implémentée sous la forme d'un EJB de type stateless.

Exemple :

```
package com.jmd.test.domaine.ejb;

import com.jmd.test.domaine.entity.Personne;
import java.util.List;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

@Stateless
public class PersonneFacade implements PersonneFacadeLocal, PersonneFacadeRemote {
    @PersistenceContext
    private EntityManager em;

    public void create(Personne personne) {
        em.persist(personne);
    }

    public void edit(Personne personne) {
        em.merge(personne);
    }

    public void remove(Personne personne) {
        em.remove(em.merge(personne));
    }

    public Personne find(Object id) {
        return em.find(com.jmd.test.domaine.entity.Personne.class, id);
    }

    public List<Personne> findAll() {
        return em.createQuery("select object(o) from Personne as o").getResultList();
    }

}
```

Les fonctionnalités offertes par l'EJB sont de type CRUD.

Le fichier persistence.xml demande simplement l'utilisation de la connexion définie dans l'annuaire.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
  <persistence-unit name="EnterpriseApplication3-ejbPU" transaction-type="JTA">
    <jta-data-source>MaTestDb</jta-data-source>
    <properties/>
  </persistence-unit>
</persistence>
```

52.7.3. La création du service web

Pour permettre une meilleure séparation des rôles de chaque classe, le service web est développé dans une classe dédiée.

Exemple :

```
package com.jmd.test.services;

import com.jmd.test.domaine.ejb.PersonneFacadeLocal;
import com.jmd.test.domaine.entity.Personne;
import java.util.List;
import javax.ejb.EJB;
import javax.jws.Oneway;
import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.ejb.Stateless;

@WebService()
@Stateless()
public class PersonneWS {
    @EJB
    private PersonneFacadeLocal ejbRef;

    @WebMethod(operationName = "create")
    @Oneway
    public void create(Personne personne) {
        ejbRef.create(personne);
    }

    @WebMethod(operationName = "edit")
    @Oneway
    public void edit(Personne personne) {
        ejbRef.edit(personne);
    }

    @WebMethod(operationName = "remove")
    @Oneway
    public void remove(Personne personne) {
        ejbRef.remove(personne);
    }

    @WebMethod(operationName = "find")
    public Personne find(Object id) {
        return ejbRef.find(id);
    }

    @WebMethod(operationName = "findAll")
    public List<Personne> findAll() {
        return ejbRef.findAll();
    }

}
```

L'injection de dépendances est utilisée pour laisser le conteneur fournir au service web une référence sur l'instance de l'EJB.

52.8. L'utilisation des EJB par un client

La suite de cette section sera développée dans une version future de ce document

52.8.1. Pour un client de type application standalone

Un client distant est en mesure d'utiliser des EJB possédant une interface de type Remote : dans ce cas, plusieurs opérations sont à réaliser

- Connexion au serveur JNDI
- Recherche de l'interface Remote de l'EJB dans JNDI
- Récupération du proxy via JNDI
- Utilisation de l'EJB au travers du proxy

Les informations nécessaires à la connexion à l'annuaire JNDI du serveur d'application sont spécifiques à chaque implémentation du serveur.

Le nom de stockage de l'interface dans JNDI est aussi spécifique au serveur utilisé.

La suite de cette section sera développée dans une version future de ce document

52.8.2. Pour un client de type module Application Client Java EE

La suite de cette section sera développée dans une version future de ce document

52.9. L'injection de dépendances

Le conteneur peut être utilisé pour assurer l'injection de dépendances de certaines ressources requises par exemple un contexte de persistance ou un autre EJB.

L'injection de dépendances est réalisée au moment de l'instanciation du bean par le conteneur.

L'injection de dépendances permet de simplifier le travail des développeurs : il n'est plus nécessaire d'invoquer l'annuaire du serveur via JNDI et de caster le résultat pour obtenir une instance de la dépendance. C'est le conteneur lui-même qui va s'en charger grâce à des annotations déchargeant le développeur de l'écriture du code utilisant JNDI ou un objet de type EJBContext.

Plusieurs annotations sont définies pour mettre en oeuvre cette injection de dépendances :

- @EJB : permet d'injecter une référence vers un autre EJB
- @Ressource : injecter une dépendance vers une ressource externe : DataSources JDBC, destinations JMS (queue ou topic), ... (annotation de Java 5)
- @PersistenceContext : injecter un objet de type EntityManager
- @WebServiceRef : injecter une référence vers un service web (annotation de JAX-WS)

Les annotations d'injection de dépendances peuvent être utilisées sur des variables d'instance ou sur des méthodes de type setter.

52.9.1. L'annotation @javax.ejb.EJB

L'annotation @EJB permet de demander au conteneur d'injecter une référence sur un EJB sans avoir à faire appel explicitement à l'annuaire du serveur avec JNDI.

Exemple :

```
@EJB
private PersonneFacadeLocal ejbReference;
```

Le conteneur utilise par défaut le type de la variable pour déterminer le type de l'instance de l'EJB qui sera injectée.

Elle s'utilise sur une classe, une méthode ou une propriété :

- sur une propriété : il est possible d'annoter un objet du type de l'EJB. L'EJB injecté sera du type de l'objet annoté.
- sur une méthode : il est possible d'annoter une méthode de type setter sur la classe de l'EJB. L'EJB injecté sera du type de l'objet en paramètre de la méthode.
- sur une classe : cela permet de déclarer que l'EJB sera utilisé à l'exécution

Elle possède plusieurs attributs :

Attribut	Rôle
Class beanInterface	Nom de l'interface de l'EJB
String beanName	Nom de l'EJB (correspond à l'attribut name des annotations @Stateless et @Stateful). Par défaut, c'est le nom de la classe de l'EJB
String description	Description de l'EJB
String mappedName	Nom JNDI de l'EJB. Cet attribut n'est pas portable
String name	Nom avec lequel l'EJB sera recherché

L'injection de dépendances est réalisée entre l'assignation de l'EJBContext et le premier appel d'une méthode de l'EJB.

S'il y a une ambiguïté pour déterminer le type de l'EJB à injecter, il est possible d'utiliser les attributs beanName et mappedName de l'annotation @EJB pour désigner l'EJB concerné.

52.9.2. L'annotation @javax.annotation.Resource

L'annotation @javax.annotation.Resource permet d'injecter des instances de ressources gérées par le conteneur tel qu'une datasource JDBC ou une destination JMS (queue ou topic) par exemple.

Cette annotation est utilisable sur une classe, une méthode ou un champ. Si l'annotation est utilisée sur une méthode ou un champ, le conteneur injecte les références au moment de l'initialisation du bean.

Elle possède plusieurs attributs :

Attribut	Rôle

String name	Nom de la ressource
Class type	Type de la ressource
AuthenticationType authenticationType	Type d'authentification à utiliser pour accéder à la ressource. Les valeurs possibles sont AuthenticationType.CONTAINER et AuthenticationType.APPLICATION
boolean shareable	Indiquer si la ressource peut être partagée entre cet EJB et d'autres EJB. Ne s'applique que sur certains types de ressources
String mappedName	Nom JNDI de la ressource
String description	Description de la ressource

52.9.3. Les annotations @javax.annotation.Resources et @javax.ejb.EJBs

La suite de cette section sera développée dans une version future de ce document

52.9.4. L'annotation @javax.xml.ws.WebServiceRef

L'annotation @WebServiceRef possède plusieurs attributs :

Attribut	Rôle
String name	nom JNDI de la ressource
String wsdlLocation	URL pointantsur le WSDL du service web
Class type	Type Java
Class value	La classe du service qui doit obligatoirement hériter de javax.xml.ws.Service
String mappedName	

Pour définir les mêmes fonctionnalités dans le descripteur de déploiement, il faut utiliser le tag <service-ref>

52.10. Les intercepteurs

Un intercepteur est une méthode qui sera exécutée selon deux types d'événements :

- intercepteur pour l'invocation de méthodes métier
- intercepteur pour des événements liés au cycle de vie de l'EJB

Un intercepteur permet de définir des traitements, généralement transverses, qui seront exécutés lorsque ces événements surviendront. Leur rôle est similaire à certaines fonctionnalités de base de l'AOP (programmation orientée aspet)

Les intercepteurs sont utilisables avec des EJB Session et MessageDriven.

Les annotations dédiées, utilisées pour la mise en oeuvre des intercepteurs, sont regroupées dans le package javax.interceptor :

- AroundInvoke
- ExcludeClassInterceptors
- DefaultInterceptors
- Interceptors

52.10.1. Le développement d'un intercepteur

Un intercepteur permet de définir des traitements sous la forme de méthodes qui seront exécutées soit à l'invocation d'une méthode métier soit lors d'événements liés au cycle de vie de l'EJB. Leur rôle est similaire à certaines fonctionnalités de base proposées par l'AOP.

Un intercepteur peut être défini soit :

- dans un EJB : dans ce cas il ne concerne que cet EJB
- dans une classe intercepteur : dans ce cas, il pourra être utilisé par tous les EJB qui en feront la demande en utilisant l'annotations @javax.interceptor.Interceptors

La signature des méthodes de callback diffère selon la nature de l'intercepteur :

- dans la classe d'un EJB : la signature est void nomMethode()
- dans la classe d'un intercepteur : la signature est void nomMethode(InvocationContext)

52.10.1.1. L'interface InvocationContext

L'interface javax.interceptor.InvocationContext définit les fonctionnalités pour permettre d'utiliser un contexte lors de l'invocation d'un ou plusieurs intercepteurs.

Cette interface définit plusieurs méthodes :

Méthode	Rôle
Object getTarget()	Renvoyer l'instance de l'EJB
Method getMethod()	Renvoyer la méthode métier de l'EJB qui a provoqué l'invocation de l'intercepteur. Si l'invocation est liée au cycle de vie de l'EJB alors cette méthode renvoie null
Object[] getParameters()	Renvoyer un tableau des paramètres de la méthode du bean pour laquelle l'intercepteur a été invoqué
void setParameters(Object[])	Modifier les paramètres qui seront utilisés pour l'invocation de la méthode
Map<String,Object> getContextData()	Obtenir une collection des données associées à l'invocation du callback
Object proceed()	Invoquer le prochain intercepteur de la chaîne ou de la méthode métier de l'EJB si tous les intercepteurs ont été invoqués

Une instance de type InvocationContext est passée en paramètre des intercepteurs.

Il est ainsi possible d'échanger des données entre les invocations des intercepteurs définis pour une même méthode.

Attention : une instance d'InvocationContext n'est pas partagée entre un intercepteur pour des méthodes metier et un intercepteur pour des événements liés au cycle de vie des EJB.

52.10.1.2. La définition d'un intercepteur lié aux méthodes métier

L'annotation @AroundInvoke permet de marquer une méthode qui sera exécutée lors de l'invocation des méthodes métier d'un EJB. Cette annotation ne peut être utilisée qu'une seule fois dans une même classe d'un intercepteur ou un EJB.

Une classe de type intercepteur ou une classe d'un EJB ne peuvent avoir qu'une seule méthode annotée avec l'annotation @AroundInvoke. Il n'est pas possible d'annoter une méthode métier avec l'annotation @AroundInvoke.

La signature d'une méthode annotée avec @AroundInvoke doit être de la forme :

```
Object nomMethode(InvocationContext) throws Exception
```

Une méthode annotée avec @AroundInvoke doit toujours invoquer la méthode proceed() de l'instance de type InvocationContext fournie en paramètre pour permettre l'invocation d'éventuels autres intercepteurs assossiés à la méthode.

Exemple :

```
package com.jmd.test.domaine.ejb;

import javax.interceptor.AroundInvoke;
import javax.interceptor.InvocationContext;

/**
 * Intercepteur qui calcule le temps d'exécution d'une méthode métier
 * @author jmd
 */
public class MesurePerfIntercepteur {

    @AroundInvoke
    public Object mesurerPerformance(InvocationContext ic) throws
            Exception {
        long debutExec = System.currentTimeMillis();
        try {
            return ic.proceed();
        } finally {
            long tempsExec = System.currentTimeMillis() - debutExec;
            System.out.println("[PERF] Temps d'execution de la methode " + ic.getClass()
              + "." + ic.getMethod() + " : " + tempsExec + " ms");
        }
    }
}
```

L'exemple ci-dessus permet de définir un intercepteur qui logguera le temps d'exécution des méthodes métiers des EJB.

52.10.1.3. La définition d'un intercepteur lié au cycle de vie

Un intercepteur peut être exécuté lorsques certains événements liés au cycle de vie de l'EJB surviennent, tels que la création, la destruction, la passivation ou la réactivation.

Les annotations liées au cycle de vie permettent de définir un intercepteur qui sera exécuté lorsque ces événements liés au cycle de vie de l'EJB surviendront.

Les EJB 2.x imposaient l'implémentation de méthodes d'une interface telles que ejbCreate(), ejbPassivate(), ... Avec les EJB 3.x, ces méthodes peuvent avoir un nom quelconque du moment qu'elles soient annotées avec une annotation liée à un événement du cycle de vie de l'EJB. Les annotations pour définir des callbacks sur des invocations de méthodes liées au cycle de vie de l'EJB sont :

- @javax.annotation.PostConstruct : méthode invoquée par le conteneur lorsqu'il a terminé les injections de dépendances pour un EJB et avant le premier appel à une de ces méthodes métier
- @javax.annotation.PreDestroy : méthode invoquée par le conteneur juste avant que l'EJB ne soit définitivement détruit

* @javax.ejb.PrePassivate : méthode invoquée par le conteneur lorsqu'un EJB de type session stateful va être rendu inactif (EJB session de type stateful uniquement)
* @javax.ejb.PostActivate : méthode invoquée par le conteneur lorsqu'un EJB de type session stateful va être réactivé (EJB session de type stateful uniquement)

Il est possible dans une même classe d'utiliser plusieurs de ces annotations mais il n'est pas possible d'utiliser plusieurs fois la même dans une même classe.

Une méthode annotée avec une annotation liée au cycle de vie dans une classe d'un intercepteur doit invoquer la méthode proceed() de l'instance de type InvocationContext fournie en paramètre pour permettre l'invocation des traitements liés à l'état courant du cycle de vie de l'EJB.

52.10.1.4. La mise oeuvre d'une classe d'un intercepteur

Une classe d'intercepteurs est un simple POJO qui doit obligatoirement avoir un constructeur sans paramètre et dont certaines méthodes sont annotées avec l'annotation @AroundInvoke ou avec une annotation liée au cycle de vie de l'EJB.

Exemple :

```
package com.jmd.test.domaine.ejb;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import javax.interceptor.AroundInvoke;
import javax.interceptor.InvocationContext;

public class MonIntercepteur {

    @AroundInvoke
    public Object audit(InvocationContext ic)
            throws Exception {
        System.out.println("MonIntercepteur Invocation de la methode : " + ic.getMethod());
        return ic.proceed();
    }

    @PreDestroy
    public void preDestroy(InvocationContext ic) {
        System.out.println("MonIntercepteur suppression du bean : " + ic.getTarget());
    }

    @PostConstruct
    public void postConstruct(InvocationContext ic) {
        System.out.println("MonIntercepteur Creation du bean : " + ic.getTarget());
    }
}
```

Les intercepteurs peuvent avoir accès par injection de dépendances aux ressources gérées par le conteneur (EJB, EntityManager, destination JMS, ...).

Un intercepteur métier peut lever une exception applicative puisque les méthodes métier peuvent lever une exception dans leur clause throws.

Les intercepteurs définis dans la classe de l'EJB sont exécutés après les intercepteurs précisés par l'annotation @Interceptors.

52.10.2. Les intercepteurs par défaut

Il est possible de définir des intercepteurs par défaut qui seront appliqués à tous les EJB d'un même jar.

La définition d'un intercepteur par défaut ne peut se faire que dans le descripteur de déploiement ejb-jar.xml. Ils ne peuvent pas être définis par des annotations.

Pour déclarer un intercepteur par défaut, il faut modifier le descripteur de déploiement ejb-jar.xml en utilisant un tag <interceptor-binding> fils du tag <assembly-descriptor>.

Le tag <interceptor-binding> peut avoir deux tags fils :

- <ejb-name> dont la valeur précise un filtre sur les ejb-name qui indique les EJB concernés. La valeur * permet d'indiquer que tous les EJB sont concernés.
- <interceptor-class> permet de préciser la classe pleinement qualifiée de l'intercepteur

L'intercepteur doit être déclaré dans un tag <interceptor> fils du tag <interceptors>. Le tag fils <interceptor-class> permet de préciser le nom pleinement qualifié de la classe de l'intercepteur.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar xmlns = "http://java.sun.com/xml/ns/javaee"
         version = "3.0"
         xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation = "http://java.sun.com/xml/ns/javaee
           http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd">
  <interceptors>
    <interceptor>
      <interceptor-class>com.jmd.test.domaine.ejb.MesurePerfIntercepteur</interceptor-class>
    </interceptor>
  </interceptors>

  <assembly-descriptor>
    <interceptor-binding>
      <ejb-name>*</ejb-name>
      <interceptor-class>com.jmd.test.domaine.ejb.MesurePerfIntercepteur</interceptor-class>
    </interceptor-binding>
  </assembly-descriptor>
</ejb-jar>
```

Les intercepteurs par défaut sont toujours invoqués avant les autres intercepteurs.

Pour empêcher l'invocation d'un intercepteur par défaut pour un EJB, il faut l'annoter avec l'annotation @javax.interceptor.excludeDefaultInterceptors.

Les intercepteurs offrent donc deux avantages :

- ils peuvent s'appliquer à tout ou partie des EJB contenus dans le même jar que l'intercepteur
- la description de leur application est centralisée et configurable dans le descripteur de déploiement ce qui permet facilement de la modifier (exemple : désactiver un intercepteur par défaut)

52.10.3. Les annotations des intercepteurs

Plusieurs annotations peuvent être utilisées lors de la mise en oeuvre des intercepteurs. Celles-ci sont soit des annotations dédiées contenues dans le package javax.interceptor soit des annotations standards de l'API Java contenues dans le package javax.annotation.

52.10.3.1. L'annotation @javax.annotation.PostContruct

L'annotation @javax.annotation.Postconstruct permet de définir un intercepteur qui est lié à l'événement de création du cycle de vie de l'EJB.

La méthode annotée avec cette annotation sera invoquée par le conteneur après l'initialisation de l'EJB et l'injection des dépendances et avant l'appel de la première méthode.

Elle peut être utilisée dans la classe d'un intercepteur ou dans la classe d'un EJB mais dans les deux cas, une seule méthode peut être annotée avec cette annotation.

Elle s'utilise sur une méthode dont la signature doit respecter quelques contraintes :

- elle ne doit pas avoir de valeur de retour
- elle ne peut pas lever d'exception de type checked
- elle ne peut pas être ni static ni final
- elle ne possède aucun attribut.

52.10.3.2. L'annotation @javax.annotation.PreDestroy

L'annotation @javax.annotation.PreDestroy permet de définir un intercepteur qui est lié à l'événement de suppression du cycle de vie de l'EJB.

La méthode annotée avec cette annotation sera invoquée par le conteneur avant que l'EJB ne soit détruit du conteneur. Celle-ci pourra par exemple procéder à la libération de ressources.

Elle peut être utilisée dans la classe d'un intercepteur ou dans la classe d'un EJB mais dans les deux cas, une seule méthode peut être annotée avec cette annotation.

Elle s'utilise sur une méthode dont la signature doit respecter quelques contraintes :

- elle ne doit pas avoir de valeur de retour
- elle ne peut pas lever d'exception de type checked
- elle ne peut pas être ni static ni final
- elle ne possède aucun attribut.

52.10.3.3. L'annotation @javax.interceptor.AroundInvoke

L'annotation @javax.interceptor.AroundInvoke permet de définir un intercepteur qui est lié à l'exécution de méthodes métier. Cette annotation peut être utilisée dans la classe d'un intercepteur ou dans la classe d'un EJB mais dans les deux cas, une seule méthode peut être annotée avec cette annotation.

Elle s'utilise sur une méthode. Elle ne possède aucun attribut.

52.10.3.4. L'annotation @javax.interceptor.ExcludeClassInterceptors

L'annotation @javax.interceptor.ExcludeClassInterceptors permet de demander d'inhiber l'invocation des intercepteurs pour une méthode. L'inhibition ne concerne pas les intercepteurs par défaut.

Elle s'utilise sur une méthode. Elle ne possède aucun attribut

52.10.3.5. L'annotation @javax.interceptor.ExcludeDefaultInterceptors

L'annotation @javax.interceptor.ExcludeDefaultInterceptors permet d'inhiber l'invocation des intercepteurs par défaut. Utilisée sur la classe d'un bean, cette annotation inhibe l'invocation des intercepteurs par défaut pour toutes les méthodes métier du bean. Utilisée sur une méthode d'un bean, cette annotation inhibe l'invocation des intercepteurs par défaut pour cette méthode.

Elle s'utilise sur une classe ou une méthode.

52.10.3.6. L'annotation @javax.interceptor.Interceptors

L'annotation @javax.interceptor.Interceptors permet de définir les classes d'intercepteurs qui seront invoquées par le conteneur. Si plusieurs classes d'intercepteurs sont définies alors elles seront invoquées dans l'ordre de leur définition dans l'annotation.

Si l'annotation est utilisée sur la classe du bean alors les intercepteurs seront invoqués pour chaque méthode du bean. Si l'annotation est utilisée sur une méthode alors les intercepteurs seront invoqués uniquement pour la méthode.

Les intercepteurs sont invoqués dans un ordre précis :

- les intercepteurs par défaut
- les intercepteurs au niveau classe
- les intercepteurs au niveau méthode

Elle s'utilise sur une classe ou une méthode. Elle possède un seul attribut :

Attribut	Rôle
Class[] value	Préciser un tableau de classes d'intercepteurs. L'ordre des intercepteurs dans le tableau définit leur ordre d'invocation (obligatoire)

52.10.4. L'utilisation d'un intercepteur

Chaque EJB qui souhaite utiliser un intercepteur devra l'ajouter via l'annotation @javax.interceptor.Interceptors.

Exemple :

```
package com.jmd.test.domaine.ejb;

import com.jmd.test.domaine.entity.Personne;
import java.util.List;
import javax.ejb.Stateless;
import javax.interceptor.Interceptors;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

@Stateless
@Interceptors({ com.jmd.test.domaine.ejb.MonIntercepteur.class})
public class PersonneFacade implements PersonneFacadeLocal, PersonneFacadeRemote {
    @PersistenceContext
    private EntityManager em;

    public void create(Personne personne) {
        em.persist(personne);
    }

...

    public List<Personne> findAll() {
        return em.createQuery("select object(o) from Personne as o").getResultList();
    }

}
```

Lors du lancement du serveur d'applications et de l'appel de cet EJB, les traces suivantes sont affichées dans la console du serveur d'applications :

Résultat :

```
...
Creation du bean
: com.jmd.test.domaine.ejb.PersonneFacade@1697e2a
...
```

```
Invocation de la
methode : public java.util.List
com.jmd.test.domaine.ejb.PersonneFacade.findAll()
...
```

Plusieurs intercepteurs peuvent être indiqués via cette annotation : leur ordre d'exécution sera celui dans lequel ils sont précisés dans l'annotation.

Un intercepteur est toujours exécuté dans la même transaction et le même contexte de sécurité que la méthode qui est à l'origine de son invocation.

Par défaut, si l'intercepteur est défini au niveau de la classe de l'EJB, toutes les méthodes concernées de l'EJB provoqueront l'invocation de l'intercepteur par le conteneur.

Si l'intercepteur est défini au niveau d'une méthode, l'intercepteur ne sera exécuté qu'à l'invocation de la méthode annotée.

L'annotation @javax.interceptor.ExcludeClassInterceptors sur une méthode permet de demander que l'exécution des intercepteurs de type @AroundInvoke précisés dans l'annotation @Interceptors soit ignorée pour la méthode.

L'annotation @javax.interceptorExcludeDefaultInterceptors sur une classe ou une méthode permet de demander que l'exécution des intercepteurs par défaut soit ignorée.

Il est possible d'associer un intercepteur à un EJB dans le descripteur de déploiement, donc sans utiliser l'annotation @Interceptors.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar xmlns = "http://java.sun.com/xml/ns/javaee"
         version = "3.0"
         xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation = "http://java.sun.com/xml/ns/javaee
           http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd">
  <interceptors>
    <interceptor>
      <interceptor-class>com.jmd.test.domaine.ejb.MesurePerfIntercepteur</interceptor-class>
    </interceptor>
  </interceptors>

  <assembly-descriptor>
    <interceptor-binding>
      <ejb-name>PersonneFacade</ejb-name>
      <interceptor-class>com.jmd.test.domaine.ejb.MesurePerfIntercepteur</interceptor-class>
    </interceptor-binding>
  </assembly-descriptor>
</ejb-jar>
```

52.11. Les EJB de type MessageDriven

Les EJB de type MessageDriven permettent de réaliser des traitements asynchrones exécutés à la réception d'un message dans une queue JMS.

Ils ne proposent pas d'interface locale ou distante et ne peuvent pas être utilisés via un service web. Pour connecter le bean à une queue JMS, il faut que le bean implémente l'interface javax.jms.MessageListener.

Cette interface définit la méthode onMessage(Message).

52.11.1. L'annotation @ javax.ejb.MessageDriven

L'annotation @ javax.ejb.MessageDriven permet de préciser qu'un EJB est de type MessageDriven. Elle s'utilise sur une classe qui encapsule un EJB.

L'annotation @MessageDriven possède plusieurs attributs optionnels :

Attribut	Rôle
ActivationConfigProperty[] activationConfig	Préciser les informations de configuration (type de endpoint, destination (queue ou topic), mode d'aquittement des messages, ...) sous la forme d'un tableau d'annotations de type @javac.ejb.ActivationConfigProperty (optionnel)
String description	Description de l'EJB (optionnel)
String mappedName	Nom sous lequel l'EJB sera mappé. Peut aussi être utilisé pour désigner le nom JNDI de la destination utilisée (optionnel)
Class messageListenerInterface	Préciser l'interface de type message Listener. Il faut utiliser cet attribut si l'EJB n'implémente pas d'interface ou implémente plusieurs interfaces différentes de java.io.Serializable, java.io.Externalizable, ou une ou plusieurs interfaces du package javax.ejb. La valeur par défaut est Object.class (optionnel)
String name	Nom de l'EJB. La valeur par défaut c'est le nom non qualifié de la classe (optionnel)

52.11.2. L'annotation @javax.ejb.ActivationConfigProperty

Les paramètres nécessaires à la configuration de l'EJB notamment le type et la destination sur laquelle le bean doit écouter doivent être précisés grâce à l'attribut activationConfig. Cet attribut est un tableau d'objets de type ActivationConfigProperty.

L'annotation @javax.ejb.ActivationConfigProperty permet de préciser le nom et la valeur d'une propriété de la configuration des EJB de type MessageDriven. Elle s'utilise dans la propriété activationConfig d'une annotation de type javax.ejb.MessageDriven.

Elle possède plusieurs attributs :

Attribut	Rôle
String propertyName	Préciser le nom de la propriété (obligatoire)
String propertyValue	Préciser la valeur de la propriété (obligatoire)

Exemple :

```
@MessageDriven(mappedName = "jms/MonEJBQueue", activationConfig = {
  @ActivationConfigProperty(propertyName="acknowledgeMode", propertyValue="Auto-acknowledge"),
  @ActivationConfigProperty(propertyName="destinationType", propertyValue="javax.jms.Queue")
})
```

52.11.3. Un exemple d'EJB de type MDB

L'exemple ci-dessous est un EJB de type Message Driven qui écoute sur une queue nommée jms/MonEJBQueue et qui affiche sur la console le contenu des messages de type texte reçus dans la queue.

Exemple :

```
package com.jmd.test.domaine.ejb;

import javax.annotation.Resource;
```

```
import javax.ejb.ActivationConfigProperty;
import javax.ejb.MessageDriven;
import javax.ejb.MessageDrivenContext;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.TextMessage;

@MessageDriven(mappedName = "jms/MonEJBQueue", activationConfig = {
  @ActivationConfigProperty(propertyName="acknowledgeMode", propertyValue="Auto-acknowledge"),
  @ActivationConfigProperty(propertyName="destinationType", propertyValue="javax.jms.Queue")
})
public class MonEJBMessageBean implements MessageListener {

    @Resource
    private MessageDrivenContext mdc;

    public MonEJBMessageBean() {
    }

    public void onMessage(Message message) {

        TextMessage msg = null;
        try {
            if (message instanceof TextMessage) {
                msg = (TextMessage) message;
                System.out.println("Message recu = " + msg.getText());
            }
        } catch (JMSException e) {
            e.printStackTrace();
            mdc.setRollbackOnly();
        } catch (Throwable te) {
            te.printStackTrace();
        }
    }
}
```

Il est possible d'écrire un client de test par exemple sous la forme d'une servlet. Cette servlet peut utiliser l'injection de dépendances si elle s'exécute dans le même serveur d'applications

Exemple :

```
package com.jmdoudoux.test.servlet;

import java.io.*;
import java.net.*;

import javax.annotation.Resource;
import javax.jms.JMSException;
import javax.jms.MessageProducer;
import javax.jms.Queue;
import javax.jms.QueueConnection;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueSession;
import javax.jms.TextMessage;
import javax.servlet.*;
import javax.servlet.http.*;

public class EnvoyerMessage extends HttpServlet {

    @Resource(mappedName = "jms/MonEJBQueue")
    Queue queue = null;

    @Resource(mappedName = "jms/MonEJBQueueFactory")
    QueueConnectionFactory factory = null;

    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {
```

```
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet EnvoyerMessage</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<h1>Servlet EnvoyerMessage</h1>");
            out.println("<form>");
            out.println("Message : <input type='text' name='msg'><br/>");
            out.println("<input type='submit'><br/>");
            out.println("</form>");
            out.println("</body>");
            out.println("</html>");

            String msg = request.getParameter("msg");

            if (msg != null) {
                QueueConnection connection = null;
                QueueSession session = null;
                MessageProducer messageProducer = null;
                try {
                    connection = factory.createQueueConnection();

                    session = connection.createQueueSession(false,
                            QueueSession.AUTO_ACKNOWLEDGE);
                    messageProducer = session.createProducer(queue);

                    TextMessage message = session.createTextMessage();
                    message.setText(msg);
                    messageProducer.send(message);
                    messageProducer.close();
                    connection.close();

                } catch (JMSException ex) {
                    ex.printStackTrace();
                }
            }

        } finally {
            out.close();
        }
    }

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        processRequest(request, response);
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        processRequest(request, response);
    }

    public String getServletInfo() {
        return "Envoi d'un message pour test EJB de type MDB";
    }
}
```

Si le client ne s'exécute pas dans le même serveur d'application, il faut utiliser JNDI pour obtenir la queue et la fabrique de connexion.

Exemple :

```
package com.jmdoudoux.test.servlet;

import java.io.*;
import java.net.*;

import javax.annotation.Resource;
import javax.jms.JMSException;
import javax.jms.MessageProducer;
import javax.jms.Queue;
```

```
import javax.jms.QueueConnection;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueSession;
import javax.jms.TextMessage;
import javax.servlet.*;
import javax.servlet.http.*;

public class EnvoyerMessage extends HttpServlet {

    Queue queue = null;
    QueueConnectionFactory factory = null;

    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet EnvoyerMessage</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<h1>Servlet EnvoyerMessage</h1>");
            out.println("<form>");
            out.println("Message : <input type='text' name='msg'><br/>");
            out.println("<input type='submit'><br/>");
            out.println("</form>");
            out.println("</body>");
            out.println("</html>");

            String msg = request.getParameter("msg");

            if (msg != null) {


                QueueConnection connection = null;
                QueueSession session = null;
                MessageProducer messageProducer = null;
                try {
                    InitialContext ctx = new InitialContext();
                    queue = (Queue) ctx.lookup("jms/MonEJBQueue");
                    factory = (QueueConnectionFactory) ctx.lookup("jms/MonEJBQueueFactory");

                    connection = factory.createQueueConnection();
                    session = connection.createQueueSession(false,
                            QueueSession.AUTO_ACKNOWLEDGE);
                    messageProducer = session.createProducer(queue);

                    TextMessage message = session.createTextMessage();
                    message.setText(msg);
                    messageProducer.send(message);
                    messageProducer.close();
                    connection.close();

                } catch (JMSException ex) {
                    ex.printStackTrace();
                } catch (NamingException ex) {
                    ex.printStackTrace();
                }
            }

        } finally {
            out.close();
        }
    }

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        processRequest(request, response);
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        processRequest(request, response);
```

```
    }

    public String getServletInfo() {
        return "Envoi d'un message pour test EJB de type MDB";
    }
}
```

Il est important que la queue et la fabrique de connexion soient définies dans l'annuaire du serveur d'applications.

Exemple avec GlassFish : extrait du fichier sun-resources.xml

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE resources PUBLIC "-//Sun Microsystems, Inc.
  //DTD Application Server 9.0 Resource Definitions
  //EN" "http://www.sun.com/software/appserver/dtds/sun-resources_1_3.dtd">
<resources>
...
  <admin-object-resource enabled="true" jndi-name="jms/MonEJBQueue"
    object-type="user" res-adapter="jmsra" res-type="javax.jms.Queue">
    <description/>
    <property name="Name" value="PhysicalQueue"/>
  </admin-object-resource>
  <connector-resource enabled="true" jndi-name="jms/MonEJBQueueFactory"
    object-type="user" pool-name="jms/MonEJBQueueFactoryPool">
    <description/>
  </connector-resource>
...
</resources>
```

Les EJB de type Message Driven peuvent exploiter toutes les fonctionnalités de JMS : utilisation d'une queue ou d'un topic comme destination, utilisation des différents types de messages (TextMessage, ObjectMessage, ...)

52.12. Le packaging des EJB

Les EJB doivent être packagés dans une archive de type jar qui doit contenir tous les éléments nécessaires à l'exécution des EJB qu'elle contient.

Le fichier jar peut lui-même être incorporé dans une archive de type EAR (Enterprise Archive) qui peut regrouper plusieurs modules dans une même entité (EJB, application web, application client lourde, ...).

La suite de cette section sera développée dans une version future de ce document

52.13. Les tests des EJB

La suite de cette section sera développée dans une version future de ce document

52.14. Les transactions

Une transaction exécute plusieurs unités de traitements qui utilisent une ou plusieurs ressources généralement une base de données. Ces unités de traitements forment un ensemble d'activités qui interagit pour former un tout fonctionnel : leurs exécutions doivent toutes réussir ou aucune ne doit être exécutée.

Le but d'une transaction est de s'assurer que toutes les unités de traitements qu'elles incluent seront toutes correctement exécutées ou qu'aucune ne sera exécutée si un problème survient.

Une transaction permet d'assurer l'intégrité des données car soit elle s'exécute correctement dans son intégralité soit elle ne fait aucune modification.

Une transaction possède quatre caractéristiques connues sous l'acronyme ACID :

- Atomic : l'exécution doit être correcte dans son intégralité ou ne pas avoir lieu. Chaque unité de traitement doit être exécutée sans erreur : si une erreur survient alors toutes les modifications réalisées dans les précédentes unités d'exécution doivent être annulées pour revenir à l'état initial avant le début de la transaction
- Consistent : le développeur doit s'assurer que les modifications réalisées dans une transaction doivent être consistantes. Par exemple, lors d'une opération bancaire de transfert de fond entre deux comptes, le montant du débit et du crédit sur chacun des comptes doit être identique
- Isolated : les données mises à jour dans la transaction ne doivent pas être modifiées en dehors de la transaction durant son execution
- Durable : le résultat de l'exécution correcte de la transaction doit être rendu persistant

L'abandon d'une transaction ne doit donc pas simplement se limiter à son arrêt, il est aussi obligatoire d'annuler toutes les mises à jour déjà réalisées par la transaction pour permettre de laisser le système dans son état initial au lancement de la transaction.

52.14.1. La mise en oeuvre des transactions dans les EJB

Le conteneur d'EJB propose un support des transactions par déclaration ce qui évite d'avoir à mettre en oeuvre explicitement une API de gestion des transactions dans le code.

Dans les EJB, une transaction concerne une méthode d'un EJB : cette transaction inclus tous les traitements contenus dans la méthode. Ceci inclus donc aussi les appels aux méthodes d'autres EJB sous réserve de leur déclaration de participation à une transaction.

La transaction est aussi propagée au contexte de persistance assuré par les EntityManager. Si la transaction est validée, alors le contexte de persistance va rendre persistante les modifications effectuées durant la transaction.

Tous les traitements inclus dans la transaction définissent la portée de la transaction.

Lorsque la transaction est gérée par le conteneur, la décision de valider ou d'abandonner la transaction est prise par le conteneur. Une transaction est abandonnée si une exception est levée dans les traitements de la méthode ou par une des méthodes de l'EJB appelée dans ces traitements.

52.14.2. La définition de transactions

L'annotation @TransactionAttribute implémentée dans la classe javax.ejb.TransactionAttribute ou le descripteur des EJB permettent de mettre en oeuvre les transactions par déclaration. Ceci permet de modifier les caractéristiques de la transaction sans avoir à modifier le code des traitements dans la méthode de l'EJB.

52.14.2.1. La définition du mode de gestion des transactions dans un EJB

L'annotation javax.ejb.TransactionManagement permet de préciser le mode de gestion des transactions dans un EJB de type session ou message driven. Ce mode peut prendre deux valeurs :

- gestion par le container (valeur par défaut)
- gestion par le code de l'EJB

Elle s'utilise sur une classe d'un EJB session ou message driven.

Elle possède un attribut :

Attribut	Rôle
TransactionManagementType value	Préciser le mode de gestion des transactions dans l'EJB. Cet attribut peut prendre deux valeurs : • TransactionManagementType.CONTAINER (valeur par défaut) • TransactionManagementType.BEAN

Dans le cas où le mode précisé est BEAN, il est nécessaire de coder la gestion des transations dans les méthodes qui en ont besoin en utilisant l'API JTA.

52.14.2.2. La définition de transactions avec l'annotation @TransactionAttribute

L'annotation @javax.ejb.TransactionAttribute permet de préciser dans quel contexte transactionnel une méthode d'un EJB sera invoquée. Cette annotation est incompatible avec la valeur BEAN de l'annotation TransactionManagement.

Elle s'utilise sur une classe d'un EJB ou sur une méthode d'un EJB session. Utilisée sur une classe, l'annotation s'applique à toutes les méthodes de l'EJB.

Elle possède un attribut :

Attribut	Rôle
TransactionAttributeType value	Préciser le contexte transactionnel d'invocation d'une méthode de l'EJB. Cet attribut peut prendre plusieurs valeurs : • TransactionAttributeType.MANDATORY • TransactionAttributeType.REQUIRED (valeur par défaut) • TransactionAttributeType.REQUIRED_NEW • TransactionAttributeType.SUPPORTS • TransactionAttributeType.NOT_SUPPORTED • TransactionAttributeType.NEVER

L'annotation @TransactionAttribute peut prendre différentes valeurs

- NOT_SUPPORTED : suspend la propagation de la transaction aux traitements de la méthode et des appels aux autres EJB de ces traitements. Une éventuelle transaction démarrée avant l'appel d'une méthode marquée avec

cet attribut est suspendue jusqu'à la sortie de la méthode.
* SUPPORTS : la méthode est incluse dans une éventuelle transaction démarrée avant son appel. Cet attribut permet à la méthode d'être incluse ou non dans une transaction
* REQUIRED : la méthode doit obligatoirement être incluse dans une transaction. Si une transaction est démarrée avant l'appel de cette méthode, alors la méthode est incluse dans la portée de la transaction. Si aucune transaction n'est définie à l'appel de la méthode, le conteneur va créer une nouvelle transaction dont la portée concernera les traitements de la méthode et les appels aux EJB de ces traitements. La transaction prend fin à la sortie de la méthode (valeur par défaut lorsque l'annotation n'est pas utilisée ou définie dans le fichier de déploiement)
* REQUIRES_NEW : une nouvelle transaction est systématiquement démarrée même si une transaction est démarrée lors de l'appel de la méthode. Dans ce cas, la transaction existante est suspendue jusqu'à la fin de l'exécution de la méthode
* MANDATORY : la méthode doit obligatoirement être incluse dans la portée d'une transaction existante avant son appel. Aucune transaction ne sera créée et elle doit obligatoirement être fournie par le client appelant. L'appel de la méthode non incluse dans la portée d'une transaction lève une exception de type javax.ejb.EJBTransactionRequiredException
* NEVER : la méthode ne doit jamais être appelée dans la portée d'une transaction. Si c'est le cas, une exception de type EJBException est levée

Cette annotation peut être utilisée au niveau de l'EJB (dans ce cas, toutes les méthodes de l'EJB utilisent la même déclaration des attributs de transaction) ou au niveau de chaque méthode.

52.14.2.3. La définition de transactions dans le descripteur de déploiement

Les déclarations des attributs relatives aux transactions peuvent aussi être faites dans le descripteur de déploiement. Le tag <container-transaction> est utilisé pour préciser les attributs de transaction d'une ou plusieurs méthodes d'un EJB.

Le tag fils <method> permet de préciser la ou les méthodes d'un EJB concerné. Le tag fils <ejb-name> permet de préciser l'EJB. Le tag <method-name> permet de préciser la méthode concernée ou toutes les méthodes de l'EJB en mettant * comme valeur au tag.

Le tag fils <trans-attribute> permet de préciser l'attribut de transaction à utiliser.

52.14.2.4. Des recommandations sur la mise en oeuvre des transactions

Il est fortement recommandé d'utiliser un contexte de persistance (EntityManager) dans la portée d'une transaction afin de s'assurer que tous les accès à la base de données se font dans un contexte transactionnel. Ceci implique d'utiliser les attributs de transaction Required, Required_New ou Mandatory.

Un EJB de type Message Driven ne peut utiliser que les attributs de transaction NotSupported et Required. L'attribut NotSupported précise que les messages ne seront pas traités dans une transaction. L'attribut Required précise que les messages seront traités dans une transaction créée par le conteneur.

Il n'est pas possible d'utiliser l'attribut Mandatory avec un EJB qui est proposé sous la forme d'un service web.

La gestion des attributs de transaction est importante car l'utilisation d'un EJB dans un contexte transactionnel est coûteux en ressources. Il faut bien tenir compte du fait que la valeur par défaut des attributs de transaction est utilisée si aucun attribut n'est précisé et que cet attribut par défaut est REQUIRED, ce qui place automatiquement l'EJB dans un contexte transactionnel.

Il est donc fortement recommandé d'utiliser un attribut de transaction NotSupported lorsqu'aucune transaction n'est requise.

52.15. La mise en oeuvre de la sécurité

Les autorisations reposent en Java EE sur la notion de rôle. Un ou plusieurs rôles sont affectés à un utilisateur. L'attribution des autorisations se fait donc au niveau rôle et non au niveau utilisateur.

Même s'il est possible d'utiliser une API dédiée, généralement la mise en oeuvre de la sécurité dans les EJB se fait de manière déclarative.

Seuls les EJB de type Session peuvent être sécurisés.

Pour définir des restrictions, il faut utiliser le descripteur de déploiement ou les annotations dédiées. Ces restrictions reposent sur la notion de rôle.

Lorsqu'une méthode est invoquée et que le conteneur détecte une violation des restrictions d'accès alors ce dernier lève une exception de type javax.ejb.EJBAccessException qui devra être traitée par le client appelant.

52.15.1. L'authentification et l'identification de l'utilisateur

Lorsqu'un client utilise des fonctionnalités du conteneur d'EJB, il possède un identifiant de sécurité durant sa connection. L'authentification de l'utilisateur est à la charge de l'application cliente.

La façon dont l'utilisateur est fourni au conteneur est dépendante de l'implémentation des EJB utilisés. Généralement cela se fait en passant des propriétés lors de la recherche du contexte JNDI.

Certains serveurs d'applications utilisent des mécanismes plus complexes et plus riches fonctionnellement en mettant en oeuvre l'API JAAS par exemple.

Lors de l'invocation des méthodes des EJB, cet identifiant de sécurité est passé implicitement à chaque appel pour permettre au conteneur de vérifier les autorisations d'utilisation par l'utilisateur.

52.15.2. La définition des restrictions

La définition des restrictions d'accès permet la mise en oeuvre des mécanismes d'autorisation.

Lorsqu'un utilisateur invoque un EJB, le conteneur contrôle les autorisations d'exécution de la méthode de l'EJB invoquée en comparant le ou les rôles de l'utilisateur avec le ou les rôles autorisés à exécuter la méthode de l'EJB.

Le mécanisme d'autorisations est précisement défini dans les spécifications des EJB. La définition des autorisations peut se faire de façon déclarative de deux façons différentes :

- utilisation des annotations dans le code de classe des EJB
- utilisation du descripteur de déploiement

52.15.2.1. La définition des restrictions avec les annotations

Par défaut, toutes les méthodes publiques d'un EJB peuvent être invoquées sans restriction de sécurité.

La définition de restrictions d'accès à un EJB se fait principalement grâce à l'annotation @javax.annotation.security.RolesAllowed qui permet de préciser les rôles qui seront autorisés à invoquer la méthode de l'EJB.

L'annotation @RolesAllowed peut s'utiliser :

- sur la classe de l'EJB : dans ce cas, cela définit les restrictions par défaut pour toutes les méthodes de l'EJB

- sur une méthode de l'EJB : dans ce cas, cela définit les restrictions pour la méthode en remplaçant les restrictions par défaut déjà définies

L'annotation @PermitAll permet l'invoquation par tout le monde : c'est l'annotation par défaut si aucune restriction n'est définie.

L'annotation @DenyAll permet d'empêcher l'invocation d'une méthode quelque soit le rôle de l'utilisateur qui l'invoque.

L'annotation @RunAs permet de forcer le rôle sous lequel l'EJB est exécuté dans le conteneur. Cette annotation ne fait aucun contrôle d'accessibilité.

52.15.2.2. La définition des restrictions avec le descripteur de déploiement

La définition de la configuration de sécurité incluant les rôles et les restrictions d'accès peut être réalisée en tout ou partie dans le descripteur de déploiement.

Les restrictions d'accès sont définies dans un tag <method-permission>

Le tag <method-permission> peut avoir plusieurs tags fils :

- un ou plusieurs tag <role-name> qui permet de préciser un rôle autorisé à utiliser la méthode
- ou le tag <unchecked> qui est équivalent à l'annotation @PermitAll
- un tag <method> qui précise la ou les méthodes concernées

Le tag <method> possède plusieurs tags fils :

- <ejb-name> qui précise le nom de l'EJB concerné
- <method-name> qui précise la méthode ou toutes les méthodes en utilisant le caractère *. Remarque : il n'est pas possible de combiner l'utilisation de caractères avec le caractère *
- <method-params> qui est optionnel permet de déterminer les méthodes concernées en cas de surcharge. Chacun des paramètres est défini avec un tag <method-param> qui contient le type du paramètre
- <method-intf> qui est optionnel permet de préciser l'interface d'accès. Les valeurs possibles sont : Remote, Local, Home, LocalHome et ServicePoint
- <description> qui est optionnel permet de fournir une description

Pour empêcher l'accès à certaines méthodes, il faut utiliser le tag <exclude-list> fils du tag <assembly-descriptor>. Le tag <exclude-list> a un rôle équivalent à l'annotation @DenyAll. Chaque méthode concernée est décrite avec un tag fils <method>.

52.15.3. Les annotations pour la sécurité

Les spécifications des EJB 3.0 définissent plusieurs annotations pour gérer et mettre en oeuvre la sécurité dans les accès réalisés sur les EJB.

52.15.3.1. javax.annotation.security.DeclareRoles

L'annotation @DeclareRoles permet de définir la liste des rôles qui sont utilisés par un EJB pour sécuriser l'invocation de ses méthodes.

Cette annotation est utile pour préciser les rôles au conteneur dans le cas ou les restrictions d'accès sont définies par programmation. Elle peut aussi être utilisée pour fournir au conteneur explicitement la liste des rôles implicitement définis dans les annotations @RolesAllowed.

L'annotation @DeclareRoles s'applique uniquement sur une classe. Elle ne possède qu'un seul attribut :

Attribut	Rôle
String[] value	Préciser le ou les rôles utilisés lors de contrôle d'accès à l'EJB (obligatoire)

52.15.3.2. javax.annotation.security.DenyAll

Aucun client ne peut invoquer la méthode de l'EJB qui est marquée avec cette annotation.

L'annotation @DenyAll s'applique uniquement sur une méthode. Elle ne possède pas d'attribut.

52.15.3.3. javax.annotation.security.PermitAll

L'annotation @PermitAll permet de préciser que la ou les méthodes de l'EJB n'ont aucune restriction d'accès.

L'annotation @PermitAll s'applique sur une classe ou une méthode. Elle ne possède pas d'attribut.

Cette annotation est l'annotation par défaut pour un EJB si aucune restriction d'accès n'est explicitement définie.

52.15.3.4. javax.annotation.security.RolesAllowed

L'annotation @RolesAllowed permet de préciser les rôles qui seront autorisés à invoquer une ou plusieurs méthodes d'un EJB.

L'annotation @RolesAllowed s'applique sur une classe ou une méthode.

Appliquée à une classe, cette annotation définit les restrictions d'accès par défaut de toutes les méthodes de l'EJB

Appliquée à une méthode, cette annotation définit les restrictions d'accès pour la méthode en remplaçant les éventuelles restrictions par défaut.

Elle ne possède qu'un seul attribut :

Attribut	Rôle
String[] value	Préciser le ou les rôles qui peuvent invoquer la ou les méthodes (obligatoire)

52.15.3.5. javax.annotation.security.RunAs

L'annotation @RunAs permet de préciser le rôle sous lequel un EJB va être executé dans le conteneur indépendemment du rôle de l'utilisateur qui invoque l'EJB.

L'annotation @RunAs s'utilise sur la classe d'un EJB. Elle possède un seul attribut :

Attribut	Rôle
String value	Préciser le rôle sous lequel l'EJB s'exécute (obligatoire)

Cette annotation peut être utilisée sur un EJB de type Session ou Message Driven.

52.15.4. La mise oeuvre de la sécurité par programmation

L'interface EJBContext propose des fonctionnalités relatives à la mise en oeuvre de la sécurité.

La méthode javax.security.Principal getCallerPrincipal() permet de connaître l'utilisateur qui invoque l'EJB.

L'interface javax.security.Principal encapsule l'utilisateur qui invoque un EJB. Sa méthode getName() permet de connaître le nom de l'utilisateur.

La méthode boolean isCallerInRole() renvoie un booléen qui vaut true si l'utilisateur qui invoque l'EJB possède le rôle founi en paramètre.

Lorsque cette méthode est utilisée, il faut utiliser l'annotation @DeclareRoles en lui précisant en paramètre les rôles qui sont utilisés avec la méthode isCallerInRole() ou effectuer la déclaration équivalente dans le descripteur de déploiement. Ceci permet au conteneur de savoir que ces rôles sont utilisés par l'EJB.

L'utilisation de ces méthodes permet de mettre en oeuvre des fonctionnalités d'autorisation plus pointues que la simple vérification vis à vis d'un rôle.

53. Les EJB 3.1

La versions 3.1 des EJB comme la version précédente permet le développement rapide d'objets métier pour des applications distribuées, sécurisées, transactionnelles et portables.

La version 3.0 des EJB a permit de remettre à plat le modèle de développement pour le rendre très simple par rapport aux versions 2.X.

Cette nouvelle version apporte de nouvelles fonctionnalités (Les interfaces Local sont optionnelles pour les EJB Session, EJB Ssingleton, les invocations asynchrones, EJB Lite, packaging simplifié, ...) et un enrichissement de fonctionnalités existantes (Service Timer, noms JNDI portables, ...) qui permettent aux développeurs et aux architectes de répondre aux besoins de leurs applications.

Leur facilité de développement et les nouvelles fonctionnalités des EJB 3.1 leur permettent de devenir très intéressant même pour des applications de taille moyenne voir petite.

Les EJB 3.1 sont issus des spécifications de la JSR 318.

Ce chapitre contient plusieurs sections :

- ♦ Les interfaces locales sont optionnelles
- ♦ Les EJB Singleton
- ♦ EJB Lite
- ♦ La simplification du packaging
- ♦ Les améliorations du service Timer
- ♦ La standardisation des noms JNDI
- ♦ L'invocation asynchrone des EJB session
- ♦ L'invocation d'un EJB hors du conteneur

53.1. Les interfaces locales sont optionnelles

Au moins une interface (locale ou distante) est requise pour les EJB Session 3.0

Les interfaces sont un excellent moyen pour limiter le couplage et assurer la testabilité : cependant dans certains cas, elles ne sont toujours pas nécessaires notamment si les deux points précédants ne sont pas une grande préoccupation.

Avec la version 3.1, il n'est plus nécessaire de définir une interface Local pour les EJB session : la classe de l'EJB session peut être directement annotée avec @Stateless ou @Stateful.

Rendre les interfaces pour les EJB session optionnelles permet à un EJB session d'être un simple POJO.

Exemple :

```
@Stateless
public class MonEJBBean {
}
```

Les EJB session n'ont plus l'obligation de définir explicitement une interface Local : le conteneur peut simplement utiliser le bean qui par défaut expose toutes les méthodes publiques de la classe et de ses classes mères. Un client peut obtenir une référence sur ce bean en utilisant l'injection de dépendance ou une recherche dans l'annuaire JNDI comme pour les interfaces Local ou Remote.

Contrairement aux interface Local et Remote avec lesquelles la référence obtenue est du type de leur interface respective, c'est le type du bean qui est directement obtenu en tant que référence.

L'exemple ci-dessous définit un EJB session.

Exemple :

```
package com.jmdoudoux.test.ejb31.domaine;

import javax.ejb.Stateless;

@Stateless
public class MonBean {

  public String saluer() {
      return "Bonjour";
  }
}
```

Ce bean ne définit aucune interface particulière. Pour l'utiliser, par exemple dans une servlet, il suffit d'utiliser l'injection de dépendance avec l'annotation @EJB sur un objet du type de la classe d'implémentation de l'EJB.

Exemple :

```
package com.jmdoudoux.test.ejb31.servlets;

import com.jmdoudoux.test.ejb31.domaine.MonBean;
import java.io.IOException;
import java.io.PrintWriter;
import javax.ejb.EJB;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet(name="MaServlet", urlPatterns={"/MaServlet"})
public class MaServlet extends HttpServlet {

    @EJB
    private MonBean monBean;

    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet MaServlet</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<h1>" + monBean.saluer() + "</h1>");
            out.println("</body>");
            out.println("</html>");
        } finally {
            out.close();
        }
    }

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
        processRequest(request, response);
```

```
    }

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
        processRequest(request, response);
    }

    @Override
    public String getServletInfo() {
        return "Ma servlet de test";
    }

}
```

Le fait d'utiliser le type de l'implémentation du bean comme référence impose quelques contraintes :

- Lorsqu'un EJB ne définit aucune interface (ni Local ni Remote) alors le conteneur doit proposer une vue de type no-interface
- Il ne faut pas utiliser l'opérateur new pour obtenir une référence sur le bean mais toujours l'obtenir par injection de dépendances
- Une exception de type EJBException est levée avec le message «Illegal non-business method access on no-interface view » si une méthode non publique est invoquée

Toutes les méthodes publiques du bean et de ses classes mères sont exposées dans la vue no-interface. Ceci expose donc les méthodes de gestion du cycle de vie, ce qui peut ne pas être souhaité.

Seules les interfaces Local sont optionnelles : les interfaces Remote sont toujours obligatoires.

Il ne faut cependant pas abuser de cette fonctionnalité et la réserver à des cas d'applications simples : avec les IDE, le cout de création et de maintenance d'une interface sont négligeables et cela renforce le découplage.

53.2. Les EJB Singleton

Plusieurs fournisseurs de serveur d'applications permettaient de n'avoir qu'une seule instance d'un EJB en permettant de préciser le nombre maximum d'instances à créer dans leur descripteur de déploiement. Cette solution n'est malheureusement pas portable puisque dépendante de l'implémentation du fournisseur.

La version 3.1 des EJB propose un nouveau type d'EJB Session nommé Singleton pour adresser cette problématique : il est possible de définir un EJB qui aura les caractéristiques du design pattern singleton : le conteneur garantie qu'une seule instance de cet EJB sera utilisable et partagée dans le conteneur.

C'est un nouveau composant qui ressemble à un EJB Session mais qui ne peut avoir qu'une seule instance dans un conteneur pour une application.

Un EJB singleton est utilisé principalement pour partager ou mettre en cache des données dans l'application. L'avantage des EJB Singleton c'est qu'ils offrent tous les services d'un EJB : sécurité, transaction, injection de dépendances, gestion du cycle de vie et intercepteurs, ...

Un EJB singleton se définit avec l'annotation @Singleton

Par défaut, toutes les méthodes d'un EJB Singleton sont thread-safe et transactionnelles.

Les EJB de type Singleton permettent d'ajouter de nouvelles fonctionnalités aux EJB :

- Exécution de code au lancement ou à l'arrêt de l'application
- Partage de données avec gestion des accès concurrents

Exemple :

```
package com.jmdoudoux.test.ejb31;

import java.util.HashMap;
import java.util.Map;
import javax.annotation.PostConstruct;
import javax.ejb.Singleton;
import javax.ejb.LocalBean;

@Singleton
@LocalBean
public class MonCache {

  private Map<String, Object> cache;

  @PostConstruct
  public void initialiser(){
      this.cache = new HashMap<String, Object>();
  }

  public Object get(String cle){
      return this.cache.get(cle);
  }

  public void put(String cle, Object valeur){
      this.cache.put(cle, valeur);
  }

  public void clear(){
      this.cache.clear();
  }
}
```

Le conteneur garantit qu'une seule instance sera accessible à l'application : les accès à cette instance pourront être effectués par plusieurs threads.

Il est possible d'annoter certaines méthodes pour gérer le cycle de vie notamment en utiliser les annotations @PostConstruct et @PreDestroy. Ceci peut permettre de réaliser des opérations liés au cycle de vie de l'application : ces traitements étaient uniquement réalisables avant qu'avec l'API Servlet via un ServletContextListener.

L'annotation @Startup demande l'initialisation du Singleton au lancement de l'application. Cette annotation ne permet cependant pas de préciser un ordre de lancement.

Il est cependant possible de définir un ordre de démarrage des EJB Singleton en utilisant l'annotation @DependsOn. Le conteneur garantira alors que les EJB dépendants sont démarrés avant l'EJB annoté.

Le cycle de vie d'un EJB Singleton est géré par le conteneur. Par défaut, c'est le conteneur qui décide de l'instanciation et de l'initialisation d'un EJB Singleton. L'annotation @Startup permet de demander au conteneur d'initialiser l'EJB à l'initialisation de l'application.

Exemple :

```
package com.jmdoudoux.test.ejb31;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import javax.ejb.DependsOn;
import javax.ejb.Singleton;
import javax.ejb.Startup;

@Singleton
@Startup
@DependsOn({"MonSecondBean"})
public class MonBean {

    @PostConstruct
    public void initialiser() {
        System.out.println("Initialisation MonBean");
    }
```

```
    @PreDestroy
    public void Detruire() {
        System.out.println("Destruction MonBean");
    }
}
```

Exemple :

```
package com.jmdoudoux.test.ejb31;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import javax.ejb.Singleton;
import javax.ejb.LocalBean;

@Singleton
@LocalBean
public class MonSecondBean {

    @PostConstruct
    public void initialiser() {
        System.out.println("Initialisation MonSecondBean");
    }

    @PreDestroy
    public void Detruire() {
        System.out.println("Destruction MonSecondBean");
    }

}
```

Durant l'arrêt de l'application, le conteneur va supprimer l'EJB après avoir éventuellement exécuté les méthodes marquées avec l'annotation @PreDestroy.

L'état de l'EJB est maintenu par le conteneur durant toute la durée de vie de l'application : cet état n'est pas persistant à l'arrêt de l'application ou de la JVM.

La gestion des accès concurrents peut utiliser deux stratégies :

- Container Managed Concurrency (CMC) : c'est le conteneur qui gère les accès concurrents au bean. C'est la stratégie par défaut.
- Bean Managed Concurrency (BMC) : la gestion des accès concurrents est à la charge du développeur en utilisant les fonctionnalités ou les API de la plateforme.

La stratégie est précisée par l'annotation @ConcurrencyManagement qui peut prendre deux valeurs : ConcurrencyManagementType.CONTAINER ou ConcurrencyManagementType.BEAN.

La stratégie CMC répond à la plupart des besoins : elle utilise des métadonnées pour gérer les verrous. Chaque méthode possède un verrou de type read ou write précisé par une annotation.

Un verrou de type read indique que la méthode peut être accédée par plusieurs threads en simultané. Un verrou de type write indique que la méthode ne peut être accédée que par un seul thread : les invocations des autres threads sont mis en attente jusqu'à la fin de l'exécution de la méthode et réactivés un par un.

L'annotation @Lock permet de préciser le type de verrou à utiliser : elle attend en paramètre une valeur qui peut être LockType.READ ou LockType.WRITE.

Cette annotation peut être utilisée sur une classe, une interface ou une méthode. Appliquée sur une classe, cette annotation agit comme valeur par défaut pour toutes les méthodes de la classe sauf pour les méthodes qui sont annotées avec @Lock.

Le type de verrou par défaut est write.

La stratégie BMC laisse au développeur le soin de gérer par programmation la gestion des accès concurrents en utilisant notamment les opérateurs synchronized et volatile ou en utilisant l'api contenu dans le package java.util.concurrent.

Par défaut, le temps d'attente d'un thread pour invoquer une méthode de l'EJB Singleton est infini. Il est possible de définir un timeout avec l'annotation @AccessTimeout qui permet de préciser un délai maximum d'attente en millisecondes. Si ce délai est atteint sans que l'invocation ne soit réalisée alors une exception de type ConcurrentAccessTimeoutException est levée.

Exemple :

```
package com.jmdoudoux.test.ejb31;

import javax.ejb.AccessTimeout;
import javax.ejb.ConcurrencyManagement;
import javax.ejb.ConcurrencyManagementType;
import javax.ejb.DependsOn;
import javax.ejb.Lock;
import javax.ejb.LockType;
import javax.ejb.Singleton;
import javax.ejb.Startup;

@Singleton
@Startup
@DependsOn({"MonSecondBean"})
@ConcurrencyManagement(ConcurrencyManagementType.BEAN)
@Lock(LockType.READ)
@AccessTimeout(15000)
public class MonBean {
...
}
```

La spécification ne prend pas en compte le clustering : elle n'apporte donc aucune précision sur le support des singletons dans un cluster et sauf implémentation particulière du serveur d'applications, il y aura une instance du singleton dans chaque JVM ou l'application est déployée.

Le conteneur doit maintenir actif un EJB Singleton durant la durée de vie de l'application même s'il lève une exception dans une de ses méthodes.

53.3. EJB Lite

Le but d'EJB Lite est de proposer une version légère d'un conteneur d'EJB utilisable par exemple dans une application Java SE ou un conteneur web comme Tomcat.

L'utilisation des EJB est souvent associée avec des serveurs d'applications Java EE mais il existe des conteneurs d'EJB open source qui peuvent être embarqués comme OpenEJB, EasyBeans ou Embedded JBoss. Le concept est maintenant proposé en standard avec les EJB 3.1.

Le but d'EJB Lite est de permettre de standardiser un conteneur d'EJB embarquable et utilisable avec Java SE. Ceci doit permettre, par exemple, de réaliser des tests unitaires ou d'utiliser des EJB dans des applications desktop ou dans un conteneur web.

Une application web typique n'a pas forcement besoin des EJB de type MDB, des services timer, ou de l'appel distant d'EJB. La plupart des applications utilisent des EJB session locaux, la persistance, l'injection, et les transactions. EJB Lite tente de proposer une solution à cette situation en proposant une implémentation allégée.

Les EJB Lite sont un sous-ensemble de l'API EJB qui permet une utilisation des EJB locaux en dehors d'un conteneur EJB comme dans le web profile ou une application standalone. EJB Lite propose les fonctionnalités suivantes :

- Support des EJB de type session (stateless, stateful, et singleton)
- Support des EJB avec interface local ou sans interface
- L'injection
- Les intercepteurs

* La sécurité et les transactions (Container Managed Transactions et Bean Managed Transactions)

Les fonctionnalités non prises en charge dans les EJB Lite sont :

* Les EJB 2.x
* L'invocation via RMI/IIOP
* Les session bean avec interface Remote
* Les EJB de type MDB
* Le support des endpoints pour les services web
* Le service Timer
* CMP / BMP

Le conteneur d'EJB embarqué propose donc un ensemble réduit de fonctionnalités qui permet à un client d'utiliser des EJB de type session sans avoir besoin d'un serveur d'applications Java EE.

Un conteneur embarqué doit au minimum supporter les API définies dans EJB Lite mais les fournisseurs peuvent ajouter à ce support tout ou partie des fonctionnalités des EJB 3.1.

Une API est proposée pour :

* Initialiser et exécuter le conteneur
* Obtenir le contexte du conteneur

La classe EJBContainer permet une utilisation d'un conteneur d'EJB embarqué. Elle possède plusieurs méthodes :

Méthode	Rôle
static EJBContainer createEJBContainer()	créer une nouvelle instance du conteneur et de l'initialiser
Context getContext()	renvoyer un objet de type javax.naming.Context qui permet un accès à l'annuaire pour rechercher des ressources de type EJB Session
void close()	demander l'arrêt du conteneur

Généralement, il suffit d'ajouter un ou plusieurs jar dans le classpath et d'utiliser l'API pour permettre la mise en oeuvre du conteneur EJB Lite.

Les usages possibles sont nombreux notamment intégrer le conteneur dans une application standalone ou web, faciliter l'exécution de tests, ...

L'exemple suivant utilise GlassFish V3 pour mettre en oeuvre un conteneur d'EJB embarqué dans une application standalone avec des tests unitaires de l'EJB.

L'exemple contient un EJB de type Session Stateless.

Exemple :

```
package com.jmdoudoux.test.ejb.embedded.domaine;

import java.util.logging.Level;
import java.util.logging.Logger;
import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import javax.ejb.Stateless;

@Stateless
public class MonBean {

  private static Logger logger = Logger.getLogger(MonBean.class.getName());

  public long ajouter(int a, int b) {
    return a + b;
  }
```

```
  @PostConstruct
  public void initialiser() {
    logger.log(Level.INFO, "Initialisation instance de MonBean");
  }

  @PreDestroy
  public void detruire() {
    logger.log(Level.INFO, "Destruction instance de MonBean");
  }
}
```

Pour compiler la classe, il faut ajouter deux bibliothèques au classpath du projet :

- javax.ejb.jar contenu dans le sous répertoire glassfish/modules de GlassFish (C:\Program Files\sges-v3\glassfish\modules par défaut)
- glassfish-embedded-static-shell contenu dans le sous répertoire glassfish/lib/embedded de GlassFish (C:\Program Files\sges-v3\glassfish\lib\embedded par défaut)

L'application créé une instance du conteneur embarqué d'EJB qui va rechercher et déployer les EJB contenus dans le classpath. Une instance du bean est obtenue à partir de son nom JNDI et utilisée pour invoquer la méthode ajouter().

Exemple :

```
package com.jmdoudoux.test.ejb.embedded;

import com.jmdoudoux.test.ejb.embedded.domaine.MonBean;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.ejb.embeddable.EJBContainer;
import javax.naming.Context;
import javax.naming.NamingException;

public class Main {

  public static void main(String[] args) {
    EJBContainer container = EJBContainer.createEJBContainer();
    Context context = container.getContext();
    MonBean monBean;
    try {
      monBean = (MonBean) context.lookup("java:global/bin/MonBean");
      System.out.println("3+2=" + monBean.ajouter(3, 2));
    } catch (NamingException ex) {
      Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, ex);
    }

    container.close();
  }
}
```

Résultat :

```
27 déc. 2009 22:57:58 com.sun.enterprise.v3.server.AppServerStartup run
INFO: GlassFish v3 (74.2) startup time : Embedded(2508ms) startup services(316ms) total(2824ms)
27 déc. 2009 22:57:58 org.glassfish.admin.mbeanserver.JMXStartupService$JMXConnectorsStarter
Thread run
INFO: JMXStartupService: JMXConnector system is disabled, skipping.
27 déc. 2009 22:57:58 com.sun.enterprise.transaction.JavaEETransactionManagerSimplified init
Delegates
INFO: Using com.sun.enterprise.transaction.jts.JavaEETransactionManagerJTSDelegate as the
delegate
27 déc. 2009 22:57:59 AppServerStartup run
INFO: [Thread[GlassFish Kernel Main Thread,5,main]] started
27 déc. 2009 22:57:59 com.sun.enterprise.security.SecurityLifecycle <init>
INFO: security.secmgroff
27 déc. 2009 22:57:59 com.sun.enterprise.security.SecurityLifecycle onInitialization
INFO: Security startup service called
27 déc. 2009 22:57:59 com.sun.enterprise.security.PolicyLoader loadPolicy
INFO: policy.loading
```

```
27 déc. 2009 22:57:59 com.sun.enterprise.security.auth.realm.Realm doInstantiate
INFO: Realm admin-realm of classtype com.sun.enterprise.security.auth.realm.
file.FileRealm successfully created.
27 déc. 2009 22:57:59 com.sun.enterprise.security.auth.realm.Realm doInstantiate
INFO: Realm file of classtype com.sun.enterprise.security.auth.realm.file.FileRealm
successfully created.
27 déc. 2009 22:57:59 com.sun.enterprise.security.auth.realm.Realm doInstantiate
INFO: Realm certificate of classtype com.sun.enterprise.security.auth.realm.
certificate.CertificateRealm successfully created.
27 déc. 2009 22:57:59 com.sun.enterprise.security.SecurityLifecycle onInitialization
INFO: Security service(s) started successfully....
27 déc. 2009 22:58:00 com.sun.ejb.containers.BaseContainer initializeHome
INFO: Portable JNDI names for EJB MonBean : [java:global/bin/MonBean,
java:global/bin/MonBean!com.jmdoudoux.test.ejb.embedded.domaine.MonBean]
27 déc. 2009 22:58:00 com.jmdoudoux.test.ejb.embedded.domaine.MonBean initialiser
INFO: Initialisation instance de MonBean
3+2=5
27 déc. 2009 22:58:00 com.jmdoudoux.test.ejb.embedded.domaine.MonBean detruire
INFO: Destruction instance de MonBean
27 déc. 2009 22:58:00 org.glassfish.admin.mbeanserver.JMXStartupService shutdown
INFO: JMXStartupService and JMXConnectors have been shut down.
27 déc. 2009 22:58:00 com.sun.enterprise.v3.server.AppServerStartup stop
INFO: Shutdown procedure finished
27 déc. 2009 22:58:00 AppServerStartup run
INFO: [Thread[GlassFish Kernel Main Thread,5,main]] exiting
```

Une application Java SE peut utiliser un conteneur d'EJB embarqué qui s'exécute dans la même JVM et le même classloader de la JVM de l'application.

Le test unitaire de l'EJB utilise également le conteneur d'EJB embarqué pour obtenir une instance de l'EJB et invoqué sa méthode ajouter() pour vérifier sa bonne exécution.

Exemple :

```
package com.jmdoudoux.test.ejb.embedded.domaine;

import javax.ejb.embeddable.EJBContainer;
import javax.naming.Context;
import javax.naming.NamingException;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;

public class MonBeanTest {


  private EJBContainer container;
  private Context context;
  private MonBean monBean;

  @Before
  public void setUp() throws NamingException {
    container = EJBContainer.createEJBContainer();
    context = container.getContext();
    monBean = (MonBean) context.lookup("java:global/bin/MonBean");
  }

  @After
  public void tearDown() {
    container.close();
  }

  @Test
  public void testAjouter() throws Exception {
    int a = 3;
    int b = 2;
    long attendu = 5L;
    long resultat = monBean.ajouter(a, b);
    assertEquals("", attendu, resultat);
  }
```

```
}
```

Résultat :

```
27 déc. 2009 22:55:03 com.sun.enterprise.v3.server.AppServerStartup run
********************************************************************************
INFO: GlassFish v3 (74.2) startup time : Embedded(2711ms) startup services(331ms) total(3042ms)
27 déc. 2009 22:55:03 org.glassfish.admin.mbeanserver.JMXStartupService$JMXConnectorsStarter
Thread run
INFO: JMXStartupService: JMXConnector system is disabled, skipping.
27 déc. 2009 22:55:03 com.sun.enterprise.transaction.JavaEETransactionManagerSimplified init
Delegates
INFO: Using com.sun.enterprise.transaction.jts.JavaEETransactionManagerJTSDelegate as the
delegate
27 déc. 2009 22:55:03 AppServerStartup run
INFO: [Thread[GlassFish Kernel Main Thread,5,main]] started
27 déc. 2009 22:55:04 com.sun.enterprise.security.SecurityLifecycle <init>
INFO: security.secmgroff
27 déc. 2009 22:55:04 com.sun.enterprise.security.SecurityLifecycle onInitialization
INFO: Security startup service called
27 déc. 2009 22:55:04 com.sun.enterprise.security.PolicyLoader loadPolicy
INFO: policy.loading
27 déc. 2009 22:55:04 com.sun.enterprise.security.auth.realm.Realm doInstantiate
INFO: Realm admin-realm of classtype com.sun.enterprise.security.auth.
realm.file.FileRealm successfully created.
27 déc. 2009 22:55:04 com.sun.enterprise.security.auth.realm.Realm doInstantiate
INFO: Realm file of classtype com.sun.enterprise.security.auth.realm.file.FileRealm
successfully created.
27 déc. 2009 22:55:04 com.sun.enterprise.security.auth.realm.Realm doInstantiate
INFO: Realm certificate of classtype com.sun.enterprise.security.auth.
realm.certificate.CertificateRealm successfully created.
27 déc. 2009 22:55:04 com.sun.enterprise.security.SecurityLifecycle onInitialization
INFO: Security service(s) started successfully....
27 déc. 2009 22:55:04 com.sun.ejb.containers.BaseContainer initializeHome
INFO: Portable JNDI names for EJB MonBean : [java:global/bin/MonBean,
java:global/bin/MonBean!com.jmdoudoux.test.ejb.embedded.domaine.MonBean]
27 déc. 2009 22:55:05 com.jmdoudoux.test.ejb.embedded.domaine.MonBean initialiser
INFO: Initialisation instance de MonBean
27 déc. 2009 22:55:05 com.jmdoudoux.test.ejb.embedded.domaine.MonBean detruire
INFO: Destruction instance de MonBean
27 déc. 2009 22:55:05 org.glassfish.admin.mbeanserver.JMXStartupService shutdown
INFO: JMXStartupService and JMXConnectors have been shut down.
27 déc. 2009 22:55:05 com.sun.enterprise.v3.server.AppServerStartup stop
INFO: Shutdown procedure finished
27 déc. 2009 22:55:05 AppServerStartup run
INFO: [Thread[GlassFish Kernel Main Thread,5,main]] exiting
```

Le conteneur embarqué recherche les EJB à déployer dans le classpath :

- Des EJB sous la forme de classes annotées packagées dans une archive de type jar
- Des EJB sous la forme de classes annotées
- Le fichier ejb-jar.xml dans le sous répertoire META-INF

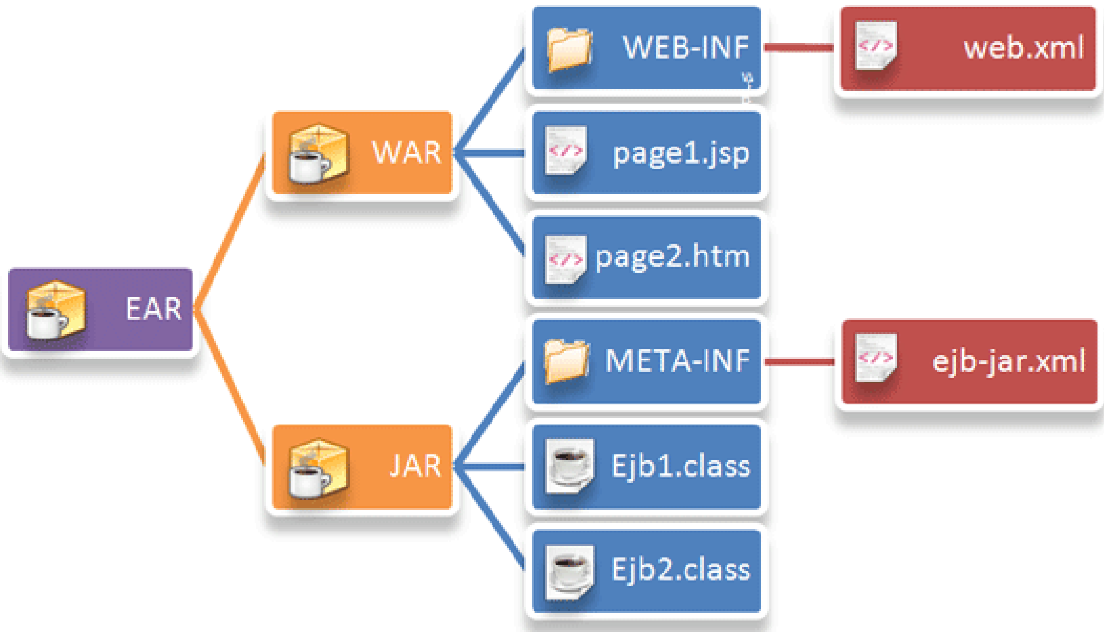
L'environnement dans lequel un EJB s'exécute est transparent pour lui : le code l'EJB est le même dans un conteneur embarqué et dans un serveur d'applications Java EE.

53.4. La simplification du packaging

Avant leur version 3.1, les EJB devaient être packagés dans une archive de type jar dédiée. Comme une application d'entreprise est généralement composée d'une partie IHM sous la forme d'une webapp packagée dans une archive de type war, il faut regrouper les archives war et jar dans une archive de type ear.

Le packaging des EJB avait été simplifié dans la version 3.0 en rendant le descripteur de déploiement optionnel. Cependant, le packaging devait toujours être fait de façon modulaire : un pour la partie web dans une archive de type war

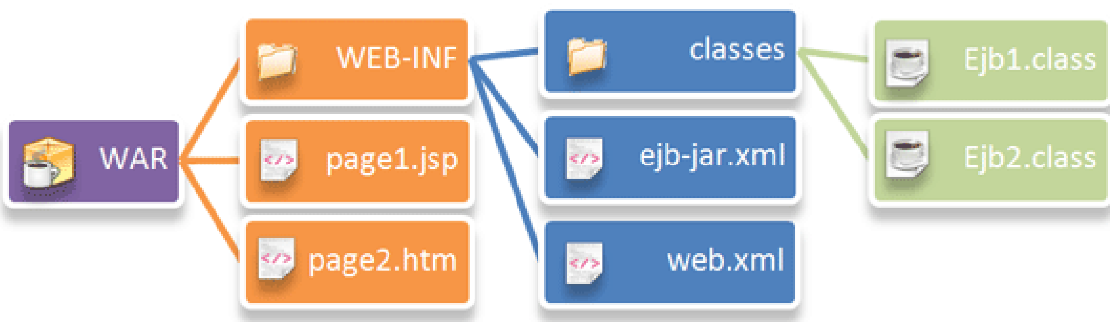
et un pour la partie EJB dans une archive de type jar, le tout regroupé dans une archive de type ear.



Ce packaging est intéressant pour rendre modulaire une grosse application mais il est complexe pour une simple application web qui utilise directement des services métiers et dont les composants n'ont pas besoin d'être partagés par plusieurs clients ou d'autres modules Java EE.

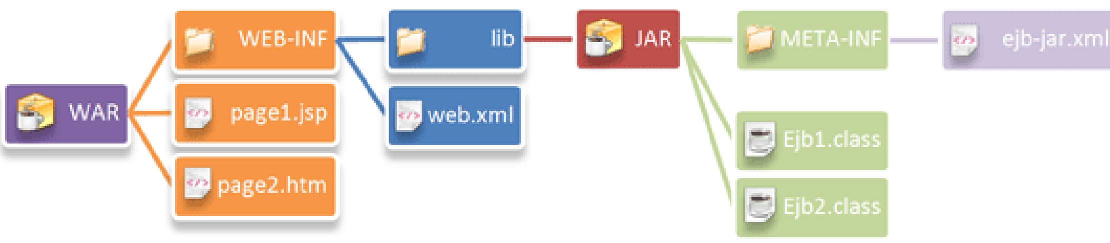
La version 3.1 propose de pouvoir intégrer les EJB directement dans la webapp sans avoir à créer un module dédié aux EJB. Les EJB qui sont des POJOs annotés peuvent être mis directement dans le sous répertoire WEB-INF/classes de la webapp et donc packagés directement dans l'archive de type war.

Si le descripteur de déploiement ejb-jar.xml doit être utilisé, il doit être placé dans le sous répertoire WEB-INF avec le fichier web.xml.



Il est aussi possible d'ajouter un jar contenant les EJB dans le sous répertoire WEB-INF/lib.

Une archive war ne peut contenir qu'un seul fichier ejb-jar.xml soit directement dans le sous répertoire WEB-INF de la webapp ou META-INF d'une des archives jar contenues dans le sous répertoire WEB-INF/lib



Comme les composants web et EJB sont packagés dans le même module, les ressources définies dans le war peuvent être partagés au travers de l'espace de nommage java:comp/env.

Ainsi, il est possible de définir une source de données dans le descripteur de déploiement web.xml et d'obtenir une référence via le contexte JNDI dans un EJB packagé dans le war.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
  <resource-ref>
    <description>Ma source de données</description>
    <res-ref-name>jdbc/bdd</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
    <res-sharing-scope>Shareable</res-sharing-scope>
  </resource-ref>
</web-app>
```

Il suffit alors de définir la source de données dans le conteneur ou le serveur d'applications dans lequel le war est déployé et d'utiliser JNDI pour obtenir une référence sur cette source de données.

Exemple :

```
package com.jmdoudoux.test.ejb31.domaine;

import java.util.logging.Level;
import java.util.logging.Logger;
import javax.ejb.Stateless;
import javax.naming.Context;

import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.sql.DataSource;

@Stateless
public class MonBean {

    public String saluer() {
        Context ctx = null;
        DataSource ds = null;
        try {
            ctx = new InitialContext();
            ds = (DataSource) ctx.lookup("java:comp/env/jdbc/bdd");
            Logger.getLogger(MonBean.class.getName()).log(Level.INFO,
              "*****"+ds.getClass().getName());
            // utilisation de la source de données
            // ...
        } catch (NamingException ex) {
            Logger.getLogger(MonBean.class.getName()).log(Level.SEVERE, null, ex);
        }

        return "Bonjour";
    }
}
```

Cette possibilité est intéressante pour intégrer des EJB dans des applications existantes.

Le nouveau modèle de déploiement pourra pleinement être utilisé avec la fonctionnalité EJB Lite qui peut être mise en oeuvre dans un simple conteneur web comme Tomcat ou Jetty. C'est d'ailleurs ce qui est proposé par le Web Profile.

Cette facilité de packaging favorise l'utilisation des EJB dans les petites et moyennes applications web.

Il est cependant recommandé de réserver ce type de packaging pour des applications simples et de conserver l'usage des archives de type ear pour des applications complexes.

53.5. Les améliorations du service Timer

Il est fréquent dans une application d'entreprise d'avoir besoin de fonctionnalités pilotées par des contraintes temporelles permettant leur déclanchement de façons régulières ou périodiques.

La version 2.1 des EJB propose le service Timer qui permet l'invocation de callbacks dans un contexte transactionnel selon des contraintes temporelles spécifiées. Ce service possède cependant quelques limitations :

- Les timers doivent être créés par programmation
- La spécification des contraintes manque de flexibilité

La version 3.1 des EJB enrichit le service timer avec :

- La possibilité de créer un timer par déclaration en utilisant l'annotation @Schedule ou le descripteur de déploiement
- L'enrichissement de l'interface TimerService pour créer un timer par programmation avec les mêmes fonctionnalités que par déclaration

Le service EJB Timer du conteneur permet de planifier l'exécution de callbacks en spécifiant un temps ou une période ou un intervalle.

Le service EJB Timer propose un support de la configuration de la planification d'un timer de deux façons :

- Soit de façon déclarative grâce à l'annotation @Schedule sur une méthode d'un EJB Session ou dans le descripteur de déploiement.
- Soit par programmation

L'annotation @Schedule met en oeuvre une expression de façon similaire à l'utilitaire cron sous Unix pour déclarer un timer qui va exécuter les traitements de la méthode qu'elle annote à chaque fois que le timer expire.

53.5.1. La définition d'un timer

Un timer peut être défini soit automatiquement par le conteneur en utilisant des annotations ou le descripteur de déploiement soit par programmation.

Les timers définis par déclaration sont automatiquement créés par le conteneur au déploiement de l'EJB.

L'annotation @Schedule s'utilise sur une méthode qui sera le callback invoqué à chaque fois que la contrainte temporelle est activée.

Les méthodes de callback invoquées lorsque le timeout d'un Timer est atteint peuvent être de deux types :

- Les méthodes associées à un Timer instanciées via une instance de TimerService
- Les méthodes annotées avec l'annotation @Schedule ou définies dans le descripteur de déploiement

Pour définir le callback d'un timer par programmation, il y a deux solutions :

- Le bean implémente l'interface javax.ejb.TimedObject
- Utiliser l'annotation @Timeout

Exemple :

```
package com.jmdoudoux.test.ejb31;

import java.text.DateFormat;
import java.util.Date;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.annotation.PostConstruct;
import javax.ejb.Timer;
import javax.annotation.Resource;
```

```
import javax.ejb.LocalBean;
import javax.ejb.ScheduleExpression;
import javax.ejb.Singleton;
import javax.ejb.Startup;
import javax.ejb.Timeout;
import javax.ejb.TimerService;

@Singleton
@Startup
@LocalBean
public class TraitementsPeriodiques3 {

  @Resource
  TimerService timerService;

  private DateFormat mediumDateFormat =
    DateFormat.getDateTimeInstance(DateFormat.MEDIUM,DateFormat.MEDIUM);

  @PostConstruct
  public void creerTimer() {
    Logger.getLogger(TraitementsPeriodiques3.class.getName()).log(Level.INFO,
      "Creation du Timer");
      ScheduleExpression scheduleExp =
        new ScheduleExpression().second("*/10").minute("*").hour("*");
      Timer timer = timerService.createCalendarTimer(scheduleExp);
  }

  @Timeout
  public void executerTraitement(Timer timer) {
    Logger.getLogger(TraitementsPeriodiques3.class.getName()).log(Level.INFO,
      "Execution du traitement toutes les 10 secondes "+mediumDateFormat.format(new Date()));
    }
}
```

Les méthodes annotées avec @Timeout ne peuvent pas lever d'exceptions.

L'interface TimedObject ne définit qu'une seule méthode ejbTimeout() qui attend en paramètre un objet de type Timer qui encapsule le timer qui invoque la méthode de callback.

Dans ce cas, une seule méthode de callback peut être définie, celle de l'interface.

Exemple :

```
package com.jmdoudoux.test.ejb31;

import java.text.DateFormat;
import java.util.Date;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.annotation.PostConstruct;
import javax.ejb.Timer;
import javax.annotation.Resource;
import javax.ejb.LocalBean;
import javax.ejb.ScheduleExpression;
import javax.ejb.Singleton;
import javax.ejb.Startup;
import javax.ejb.TimedObject;
import javax.ejb.Timeout;
import javax.ejb.TimerService;

@Singleton
@Startup
@LocalBean
public class TraitementsPeriodiques4 implements TimedObject {

  @Resource
  TimerService timerService;

  private DateFormat mediumDateFormat =
    DateFormat.getDateTimeInstance(DateFormat.MEDIUM,DateFormat.MEDIUM);
```

```
  @PostConstruct
  public void creerTimer() {
    Logger.getLogger(TraitementsPeriodiques4.class.getName()).log(Level.INFO,
      "Creation du Timer");
    ScheduleExpression scheduleExp =
      new ScheduleExpression().second("*/5").minute("*").hour("*");
    Timer timer = timerService.createCalendarTimer(scheduleExp);
  }

  public void ejbTimeout(Timer timer) {
    Logger.getLogger(TraitementsPeriodiques4.class.getName()).log(Level.INFO,
    "Execution du traitement toutes les 5 secondes "+mediumDateFormat.format(new Date()));
  }
}
```

Les méthodes de callbacks des Timers créés automatiquement sont soit annotées avec @Schedule ou @Schedules ou définis dans l'élément timeout-method du descripteur de déploiement.

Ces méthodes annotées de callbacks peuvent avoir deux signatures (où xxx est le nom de la méthode) :

- void xxx()
- ou void xxx(Timer timer)

Elles peuvent avoir n'importe quel modificateur d'accès mais ne peuvent pas être déclarées ni final ni static. Elles ne peuvent pas lever d'exception.

Comme le callback est interne à l'EJB, il ne possède aucun contexte de sécurité.

Le conteneur doit créer une nouvelle transaction si l'attribut de transaction est REQUIRED ou REQUIRED_NEW. Si la transaction échoue ou si elle est abandonnée, le conteneur doit retenter au moins une fois l'exécution du callback.

53.5.2. L'annotation @Schedule

L'annotation @Schedule permet de créer un timer dont les caractéristiques sont fournies sous la forme d'attributs de l'annotation.

La syntaxe de déclaration se fait sous la forme d'une expression dont la syntaxe est inspirée de l'outil Unix cron. Cette expression peut utiliser huit attributs :

Attribut	Valeurs possibles	Exemple
Hour	0 à 23 (heure)	hour = "23"
Minute	0 à 59 (minute)	minute = "59"
Second	0 à 59 (seconde)	second = "59"
dayOfMonth	1 à 31 : jour du mois last : dernier jour du mois -1 à -7 : nombre de jours avant la fin du mois {"1st", "2nd", "3rd", "4th", "5th", "Last"} {"Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"} : identifier un jour précis dans le mois	dayOfMonth = "1 dayOfMonth = "last" dayOfMonth = "-1" dayOfMonth = "1st Mon"
dayOfWeek	0 à 7 : jour de la semaine (0 et 7 représentent le dimanche) {"Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"}	dayOfWeek = "1"

Month	1 à 12 : le mois de l'année {"Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"} : le mois selon 3 première lettres	month = "1" month = "Jan"
Year	Une année sur 4 chiffres	year = "2010"
timezone		

La valeur fournie à chaque attribut peut prendre différentes formes :

Forme	Description	Exemple
Une valeur simple	Une valeur unique correspondant à une des valeurs possibles de l'attribut	hour = "20"
Une étoile	Représente toutes les valeurs possibles de l'attribut	dayOfMonth = "*"
Une Liste	Représente un ensemble de valeurs possibles pour l'attribut séparées par une virgule	dayOfWeek = "Mon, Wed, Thu"
Une plage	Représente une plage de valeurs consécutives possibles pour l'attribut dont les deux bornes incluses sont séparées par un tiret	year = "2010-2019"
Une incrémentation	Définie une expression de la forme x/y où la valeur est incrémentée de y dans la plage de valeurs possibles en commençant à la valeur x. Elle ne peut être appliquée que sur heure, minute et seconde. Une fois la valeur maximale atteinte, l'incrémentation s'arrête	minute= "*/10" (toutes les 10 minutes)

Les expressions possèdent des règles et des contraintes :

- La valeur par défaut des attributs hour, minute et second est 0
- La valeur par défaut des attributs dayOfWeek, dayOfMonth, month et year est « * »
- Les chaines de caractères constantes sont insensibles à la casse
- Les valeurs en double dans les listes sont ignorées

Voici quelques exemples :

Expression	Description
@Schedule(hour="6", dayOfMonth="1")	Le premier de chaque mois à 6 heure du matin
@Schedule(dayOfWeek="Mon-Fri", hour="22")	Du lundi au vendredi à 10 heure du soir
@Schedule(hour = "22", minute = "30", dayOfWeek = "Fri")	Tous les vendredi à 22 heure 30
@Schedule(hour = "10, 14, 18", dayOfWeek = "Mon-Fri")	Du lundi au vendredi à 10, 14 et 18 heure
@Schedule(hour = "*", dayOfWeek = "1")	Toutes les heures de chaque lundi
@Schedule(hour = "23", dayOfMonth = "Last Fri", month="*")	Le dernier vendredi de chaque mois à 23 heure
@Schedule(hour = "22", dayOfMonth = "-3")	Trois jours avant la fin de chaque mois à 22 heure
@Schedule(minute = "*/15", hour = "12/1")	Tous les quart d'heure à partir de midi

Exemple :

```
package com.jmdoudoux.test.ejb31;

import java.text.DateFormat;
import java.util.Date;
import java.util.logging.Level;
```

```
import java.util.logging.Logger;
import javax.ejb.LocalBean;
import javax.ejb.Schedule;
import javax.ejb.Stateless;

@Stateless
@LocalBean
public class TraitementsPeriodiques2 {

  DateFormat mediumDateFormat =
    DateFormat.getDateTimeInstance(DateFormat.MEDIUM,DateFormat.MEDIUM);

  @Schedule(dayOfWeek="Mon")
  public void traiterHebdomadaires() {
    Logger.getLogger(TraitementsPeriodiques2.class.getName()).log(Level.INFO,
      "Execution du traitement hebdomadaire");
  }

  @Schedule(minute="*/1", hour="*")
  public void traiterMinutes() {
    Logger.getLogger(TraitementsPeriodiques2.class.getName()).log(Level.INFO,
      "Execution du traitement chaque minute "+mediumDateFormat.format( new Date()));
  }


  @Schedule(second="*/30", minute="*", hour="*")
  public void traiterTrenteSecondes() {
    Logger.getLogger(TraitementsPeriodiques2.class.getName()).log(Level.INFO,
      "Execution du traitement toutes les 30 secondes "+mediumDateFormat.format(new Date()));
  }
}
```

Résultat :

```
INFO: Execution du traitement chaque minute 31 janv. 2010 16:50:00
INFO: Execution du traitement toutes les 30 secondes 31 janv. 2010 16:50:00
INFO: Execution du traitement toutes les 30 secondes 31 janv. 2010 16:50:30
INFO: Execution du traitement toutes les 30 secondes 31 janv. 2010 16:51:00
INFO: Execution du traitement chaque minute 31 janv. 2010 16:51:00
INFO: Execution du traitement toutes les 30 secondes 31 janv. 2010 16:51:30
```

L'annotation @Schredule possède un attribut info qui permet de fournir une description du timer. Ces informations peuvent être retrouvées grâce à la méthode getInfo() de l'instance de type Timer.

Il est possible d'associer plusieurs timers par déclaration à une même méthode de callback en utilisant l'annotation @Schedules qui agit comme un conteneur d'annotations @Schredule

Exemple :

```
@Schedules(
{ @Schedule(hour="20", dayOfWeek="Mon-Thu"),
  @Schedule(hour="18", dayOfWeek="Fri")
})
public void envoyerRapport() {
...
}
```

53.5.3. La persistance des timers

Un timer peut être persistant ou non : les timers non persistants ne survivent pas à un arrêt de conteneur.

La durée de vie d'un timer non persistant est lié à la durée de vie de la JVM qui l'a créé et dans lequel il s'exécute : il est considéré comme supprimé en cas d'arrêt de l'application ou arrêt volontaire ou non de la JVM.

Les timers définis par déclaration sont par défaut persistant : le conteneur les réactive automatiquement même si le conteneur est arrêté puis relancé.

Exemple : log de démarrage d'un serveur Glassfish v3

```
INFO: [TimerBeanContainer] Created  TimerBeanContainer: TimerBean
INFO: Portable JNDI names for EJB TimerBean : [java:global/ejb-timer-service-app/TimerBean,
java:global/ejb-timer-service-app/TimerBean!com.sun.ejb.containers.TimerLocal]
INFO: EJB5109:EJB Timer Service started successfully for datasource [jdbc/__TimerPool]
INFO: ==> Restoring Timers ...
INFO: <== ... Timers Restored.
INFO: Loading application ejb-timer-service-app at /ejb-timer-service-app
```

Un timer non persistant peut être créé de deux façons :

- Par déclaration : en utilisant l'attribut persistent=false de l'annotation @Schredule
- Par programmation : en utilisant la classe TimerConfig passée en paramètre de la méthode createTimer() de l'interface TimerService. Il faut fournir en paramètre de la surcharge de la méthode de création une instance de type TimerConfig pour laquelle la méthode setPersistent() a été invoquée avec la valeur false.

53.5.4. L'interface Timer

L'interface javax.ejb.Timer propose des méthodes pour annuler un timer ou obtenir des informations sur lui.

Méthode	Rôle
void cancel()	Demande la suppression du timer et de toutes ses notifications au conteneur
long getTimeRemaining()	Obtenir le nombre de millisecondes avant la prochaine notification d'expiration du Timer
Date getNextTimeout()	Obtenir la date/heure programmée de la prochaine notification d'expiration du Timer
ScheduleExpression getSchedule()	Obtenir l'objet qui définit l'expression de planification
TimerHandle getandle()	Obtenir une version sérialisable du Timer
Serializable getInfo()	Obtenir les informations complémentaires fournies lors de la création du Timer
boolean isPersistent()	Déterminer sur le timer est persistant ou non
boolean isCalendar()	Déterminer si le Timer est basé sur un calendrier

Exemple :

```
package com.jmdoudoux.test.ejb31;

import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.GregorianCalendar;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.annotation.PostConstruct;
import javax.ejb.Timer;
import javax.annotation.Resource;
import javax.ejb.LocalBean;
import javax.ejb.Singleton;
import javax.ejb.Startup;
import javax.ejb.TimedObject;
import javax.ejb.TimerConfig;
import javax.ejb.TimerService;
```

```
@Singleton
@Startup
@LocalBean
public class TraitementsPeriodiques7 implements TimedObject {

  @Resource
  TimerService timerService;
  private DateFormat mediumDateFormat =
    DateFormat.getDateTimeInstance(DateFormat.MEDIUM, DateFormat.MEDIUM);

  @PostConstruct
  public void creerTimer() {
    Logger.getLogger(TraitementsPeriodiques7.class.getName()).log(Level.INFO,
      "Creation du Timer" + mediumDateFormat.format(new Date()));

    TimerConfig config = new TimerConfig();
    config.setInfo("donnees complementaires");

    Timer timer = timerService.createSingleActionTimer(60000, config);
  }

  public void ejbTimeout(Timer timer) {
    Logger.getLogger(TraitementsPeriodiques7.class.getName()).log(Level.INFO,
      "Execution du traitement après 60s d'attente ("+timer.getInfo()+") "
      + mediumDateFormat.format(new Date()));
  }
}
```

53.5.5. L'interface TimerService

L'interface TimerService définit les méthodes pour permettre un accès au service Timer du conteneur.

Elle a été enrichie pour permettre de définir des timers par programmation.

Pour obtenir une instance de type TimerService, il faut utiliser la méthode getTimerService() de l'interface EJBContext ou demander l'injection d'une ressource de type TimerService.

L'interface TimerService propose plusieurs méthodes pour créer des instances de type Timer qui soient déclenchées de façon unique, selon un intervalle ou planifier selon un calendrier spécifié avec une expression grâce aux différentes surcharges des méthodes createTimer(), createSingleActionTimer(), createIntervalTimer() ou createCalendarTimer().

La méthode createSingleActionTimer() créé un Timer qui sera supprimé dès que son callback sera invoqué. Une version surchargée permet de préciser un délai d'attente.

Exemple :

```
package com.jmdoudoux.test.ejb31;

import java.text.DateFormat;
import java.util.Date;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.annotation.PostConstruct;
import javax.ejb.Timer;
import javax.annotation.Resource;
import javax.ejb.LocalBean;
import javax.ejb.Singleton;
import javax.ejb.Startup;
import javax.ejb.TimedObject;
import javax.ejb.TimerConfig;
import javax.ejb.TimerService;

@Singleton
@Startup
@LocalBean
public class TraitementsPeriodiques5 implements TimedObject {
```

```
  @Resource
  TimerService timerService;

  private DateFormat mediumDateFormat =
    DateFormat.getDateTimeInstance(DateFormat.MEDIUM,DateFormat.MEDIUM);

  @PostConstruct
  public void creerTimer() {
    Logger.getLogger(TraitementsPeriodiques5.class.getName()).log(Level.INFO,
      "Creation du Timer"+mediumDateFormat.format(new Date()));
    Timer timer = timerService.createSingleActionTimer(60000, new TimerConfig());
  }

  public void ejbTimeout(Timer timer) {
    Logger.getLogger(TraitementsPeriodiques5.class.getName()).log(Level.INFO,
      "Execution du traitement après 60s d'attente "+mediumDateFormat.format(new Date()));
  }
}
```

Une autre version surchargée permet de préciser une date/heure.

Exemple :

```
package com.jmdoudoux.test.ejb31;

import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.GregorianCalendar;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.annotation.PostConstruct;
import javax.ejb.Timer;
import javax.annotation.Resource;
import javax.ejb.LocalBean;
import javax.ejb.Singleton;
import javax.ejb.Startup;
import javax.ejb.TimedObject;
import javax.ejb.TimerConfig;
import javax.ejb.TimerService;

@Singleton
@Startup
@LocalBean
public class TraitementsPeriodiques6 implements TimedObject {

  @Resource
  TimerService timerService;
  private DateFormat mediumDateFormat =
    DateFormat.getDateTimeInstance(DateFormat.MEDIUM, DateFormat.MEDIUM);

  @PostConstruct
  public void creerTimer() {
    Logger.getLogger(TraitementsPeriodiques6.class.getName()).log(Level.INFO,
      "Creation du Timer" + mediumDateFormat.format(new Date()));
    GregorianCalendar calend =
      new GregorianCalendar(2010, GregorianCalendar.FEBRUARY, 7, 16, 45, 0);
    Timer timer = timerService.createSingleActionTimer(calend.getTime(), new TimerConfig());
  }

  public void ejbTimeout(Timer timer) {
    Logger.getLogger(TraitementsPeriodiques6.class.getName()).log(Level.INFO,
      "Execution du traitement" + mediumDateFormat.format(new Date()));
  }
}
```

La classe createCalendarTimer() permet de créer un Timer dont les conditions d'exécution son précisées par une instance de la classe ScheduleExpression fournie en paramètres.

La classe ScheduleExpression encapsule l'expression qui désigne le déclenchement des traitements des Timers programmés par un calendrier.

La méthode getTimers() retourne une collection des Timers associés avec l'EJB. Il est ainsi possible d'accéder à chaque Timer pour obtenir des informations ou pour les effacer.

53.6. La standardisation des noms JNDI

Tous les EJB de type Session sont enregistrés dans un annuaire avec un nom unique accessible par un client via un contexte JNDI que ce soit en utilisant directement le contexte (hors du conteneur) ou en utilisant l'injection de dépendance (dans le conteneur).

Ce nom JNDI est automatiquement définit par le conteneur à l'enregistrement de chaque EJB, chaque fournisseur utilisant sa propre nomenclature puisque les spécifications leur en laisse la latitude.

Cela pose des soucis de portabilité entre différents conteneurs. Ceci pose d'autant plus de soucis avec l'injection de dépendances puisque le conteneur doit être capable de déterminer le nom JNDI à partir des métadonnées de l'annotation @EJB.

Cette liberté laissée au fournisseur d'implémentations sur le nom JNDI sous lequel l'EJB est désigné limite la portabilité de l'application sur différents serveurs d'applications.

Ainsi, les EJB Session d'une même application déployée dans différents conteneurs se voient déployés avec un nom JNDI différents, ce qui va nécessairement poser des soucis lors des invocations par le ou les clients. Ceci va à l'encontre de la philosophie de Java EE.

La spécification standardise le nom global JNDI et deux autres espaces de nommages relatifs aux différentes portées d'une application Java EE.

La standardisation du nom JNDI par la spécification permet de définir clairement comment le nom global JNDI (global JNDI name) doit être défini, résolvant ainsi les problèmes de portabilité pour retrouver des références vers des composants ou des ressources.

Ce nom JNDI est composé de façon à le rendre unique dans une instance d'un conteneur en utilisant le préfixe java:global, le nom de l'application, le nom du module, le nom du bean et le nom de l'interface sous la forme.

java:global[/<application-name>]/<module-name>/<bean-name>!<interface-name>

Partie du nom	Description	Obligatoire
application-name	Nom de l'application dans lequel l'EJB est packagé. Par défaut c'est le nom de l'archive de type ear sans son extension sauf si le nom de l'application est précisée dans le descripteur de déploiement application.xml.	Non
module-name	Nom du module dans lequel l'EJB est packagé. Par défaut, c'est le nom de l'archive de type jar ou war sans son extension sauf si le nom du module est précisé dans le fichier ejb-jar.xml via un élément module-name	Oui
bean-name	Nom du bean Par défaut, c'est le nom de la classe d'implémentation de l'EJB sauf si le nom est précisé via l'attribut name de l'annotation @Stateless, @ Stateful et @Singleton ou via l'élément bean-name du descripteur de déploiement	Oui
interface-name	Nom pleinement qualifié de l'interface sous laquelle l'EJB est exposé. Si l'EJB ne possède aucune interface (no interface view) alors c'est le nom pleinement qualifié de la classe d'implémentation qui est utilisé.	Oui

Le nom de l'application est optionnel car il n'est connu que si l'application est packagée dans une archive de type ear.

Le nom du module est déterminé à partir de l'archive jar ou war selon le format de l'archive dans lequel l'EJB est packagé.

Le nom de l'interface n'est utile que si l'EJB implémente plusieurs interfaces (Locale et Remote) : il est inutile si l'EJB n'implémente qu'une seule interface ou aucune interface. Dans ce cas, le conteneur doit aussi associer l'EJB avec un nom JNDI court sous la forme :

java:global[/<application-name>]/<module-name>/<bean-name>

Le conteneur a aussi l'obligation d'enregistrer l'EJB dans deux autres espaces de nommage du contexte : java:app et java:module.

L'espace de nommage java:app concerne l'application. La syntaxe est la suivante :

java:app/<module-name>/<bean-name>[!<interface-name>]

L'espace de nommage java:module concerne le module. La syntaxe est la suivante :

java:module/<bean-name>[!<interface-name>]

Ceci devrait améliorer la portabilité des applications Java EE entre différents conteneurs.

53.7. L'invocation asynchrone des EJB session

L'invocation de traitements asynchrones est relativement fréquent dans les applications d'entreprises mais jusqu'à la version 3.0 incluse des EJB aucune solution standard n'était proposée pour ce besoin.

Comme les threads ne peuvent pas être utilisés dans les EJB, une façon couramment utilisée de permettre une invocation asynchrone d'un EJB est de passer par un message JMS traité par un EJB de type MDB. Cependant, le rôle principal de JMS est l'échange de messages et pas l'invocation de fonctionnalités de façon asynchrone.

De plus, cette solution n'est pas idylique car elle ne permet pas facilement d'avoir un retour à la fin des traitements réalisés.

53.7.1. L'annotation @Asynchronous

La version 3.1 des EJB propose un support pour l'invocation asynchrone des EJB de type Session en utilisant l'annotation @Asynchronous sur la méthode de l'EJB qui contient les traitements.

Cette méthode peut retourner :

- void : dans ce cas, il n'y aura aucun retour à la fin de l'exécution des traitements et la méthode ne doit lever aucune exception puisque celles-ci ne pourraient pas être traitées
- Future<T> : dans ce cas, le client pourra avoir un contrôle sur l'état de l'exécution et obtenir la valeur de retour ou une exception levée par les traitements

L'invocation asynchrone d'EJB de type Session peut être utilisée sur tous les types d'EJB Session et avec toutes les interfaces de ces EJB.

L'annotation @Asynchronous peut être utilisée sur une méthode ou une classe ou une interface.

Si l'annotation @Asynchronous est utilisée sur des méthodes alors seules ces méthodes sont invocables de façon asynchrone.

Exemple :

```
package com.jmdoudoux.test.ejb31;

import java.util.concurrent.Future;
import javax.ejb.AsyncResult;
import javax.ejb.Asynchronous;
import javax.ejb.Stateless;

@Stateless
public class MailEJB {

    @Asynchronous
    public Future<Boolean> envoyerAsync() {
       return new AsyncResult<Boolean>(true);
    }

    public Boolean envoyer() {
        return true;
    }
}
```

Si l'annotation @Asynchronous est utilisée sur la classe, toutes les méthodes exposées sont invocables de façon asynchrone.

Exemple :

```
package com.jmdoudoux.test.ejb31;

import java.util.concurrent.Future;
import javax.ejb.AsyncResult;
import javax.ejb.Asynchronous;
import javax.ejb.Stateless;

@Stateless
@Asynchronous
public class MailEJB {

    public Future<Boolean> envoyer() {
       return new AsyncResult<Boolean>(true);
    }

    public Future<Boolean> envoyerAvecCopie() {
       return new AsyncResult<Boolean>(true);
    }
}
```

Il est aussi possible d'utiliser l'annotation @Asynchronous sur une interface. Dans ce cas, les méthodes invocables de façon asynchrone seront celles précisées par l'interface puisque l'EJB sera invoqué au travers de son interface.

Exemple :

```
package com.jmdoudoux.test.ejb31;

import javax.ejb.Local;

@Local
public interface MailEJBLocal {

    Boolean envoyer();
}
```

Exemple :

```
package com.jmdoudoux.test.ejb31;

import java.util.concurrent.Future;
import javax.ejb.Asynchronous;
```

```
import javax.ejb.Remote;

@Remote
public interface MailEJBRemote {

    @Asynchronous
    Future<Boolean> envoyerAsync();
}
```

Exemple :

```
package com.jmdoudoux.test.ejb31;

import java.util.concurrent.Future;
import javax.ejb.AsyncResult;
import javax.ejb.Stateless;


@Stateless
public class MailEJB implements MailEJBRemote, MailEJBLocal {

    public Future<Boolean> envoyerAsync() {
       return new AsyncResult<Boolean>(true);
    }

    public Boolean envoyer() {
        return true;
    }
}
```

L'annotation @Asynchronous peut aussi être utilisée sur un EJB de type Singleton.

53.7.2. L'invocation d'une méthode asynchrone

Lors de l'invocation de la méthode annotée avec @Asynchronous, le client poursuit l'exécution de ces traitements sans attendre la fin de l'exécution de l'invocation.

C'est le conteneur qui garantit que les traitements seront exécutés de façon asynchrone.

L'invocation asynchrone est faite par le client (EJB, application standalone, ...) de façon transparente pour lui.

Exemple :

```
package com.jmdoudoux.test.ejb31;

import java.util.logging.Level;
import java.util.logging.Logger;
import javax.ejb.EJB;
import javax.ejb.Stateless;
import javax.jws.WebMethod;
import javax.jws.WebService;

@Stateless
@WebService
public class CommandeEJB implements CommandeEJBLocal, CommandeEJBRemote {

    @EJB
    MailEJBLocal mailEJB;

    @WebMethod
    public void valider(int id) {

        // traitement de validation de la commande
        Logger.getLogger(CommandeEJB.class.getName()).log(Level.INFO,
          "validation de la commande numero "+id);
```

```
        // envoie d'un mail de prise en compte
        mailEJB.envoyerAsync(id);

        Logger.getLogger(CommandeEJB.class.getName()).log(Level.INFO,
        "fin de la validation de la commande");
    }

}
```

La classe java.util.concurrent.Future<T>, disponible depuis la version 5 de Java SE, permet d'avoir un contrôle sur l'invocation asynchrone d'un traitement. Elle est typée avec le type de la valeur de retour à l'issu de l'exécution des traitements.

L'interface Future<V> définit plusieurs méthodes :

Méthode	Rôle
boolean cancel(boolean)	Demander une tentative d'annulation de l'exécution des traitements. Le conteneur va tenter d'annuler l'invocation si celle-ci n'a pas encore commencé. La méthode renvoie true si l'invocation a pu être annulée. Le paramètre permet de demander au conteneur d'informer le bean de la demande d'annulation si celui-ci est déjà en cours d'exécution
V get() V get(long, TimeUnit)	Renvoyer la valeur de retour des traitements Cette méthode possède deux surcharges : • Sans paramètres : attend jusqu'à la fin des traitements • Avec un timeout en paramètre : attend jusqu'à la durée du timeout puis tente de récupérer la valeur de retour
boolean isCancelled()	Préciser si l'exécution des traitements a été annulée
boolean isDone()	Préciser si l'exécution des traitements est terminée

La classe javax.ejb.AsyncResult<V> est une implémentation fournie en standard de l'interface Future<V> qui propose notamment un constructeur qui attend la valeur de retour de type V en paramètre.

La méthode wasCancelCalled() de l'interface SessionContext renvoie true si le client a invoqué la méthode Future.cancel() avec la valeur true en paramètre.

Exemple :

```
package com.jmdoudoux.test.ejb31;

import java.util.Date;
import java.util.concurrent.Future;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.annotation.Resource;
import javax.ejb.AsyncResult;
import javax.ejb.Asynchronous;
import javax.ejb.SessionContext;
import javax.ejb.Stateless;


@Stateless
public class MailEJB implements MailEJBRemote, MailEJBLocal {

   @Resource
   SessionContext ctx;

   @Asynchronous
   public Future<Boolean> envoyerAsync(int valeur) {
       boolean resultat = ( (valeur % 2) == 0);
```

```
        Logger.getLogger(MailEJB.class.getName()).log(Level.INFO,
          "debut de l'envoi du mail "+new Date());

        long i = 0;
        while ( i < 300000000 && !ctx.wasCancelCalled() ) {
            // code des traitements a executer
            i++;
        }
        if (ctx.wasCancelCalled()) {
            resultat = false;
        }

        Logger.getLogger(MailEJB.class.getName()).log(Level.INFO,
          "fin de l'envoi du mail "+new Date());

        return new AsyncResult<Boolean>(resultat);
    }

    public Boolean envoyer() {
        return true;
    }
}
```

L'exemple ci-dessus effectue un traitement qui peut être interrompu par le client. La méthode envoyerAsync() renvoie un booléen qui indique le succès des traitements : elle renvoie false si l'id fournie est impaire ou si les traitements ont été interrompus par le client.

La méthode get() de l'interface Future peut lever une exception de type ExecutionException qui va encapsuler une éventuelle exception levée par la méthode exécutée de façon asynchrone. L'exception originale est chaînée et donc accessible en utilisant la méthode getCause().

Exemple :

```
...
    @Action
    public void invocationOk() {
        executerTraitement(2, false);
    }

    @Action
    public void InvocationKo() {
        executerTraitement(1, false);
    }

    @Action
    public void invocationCancel() {
        executerTraitement(2, true);
    }

    public void executerTraitement(int valeur, boolean arret) {
        try {
            Future<Boolean> future = bean.envoyerAsync(valeur);
            if (arret) {
                Thread.sleep(1000);
                future.cancel(true);
            }
            Boolean resultat = future.get();
            jTextArea1.setText("resultat="+resultat+
              " isCancelled="+future.isCancelled());
            Logger.getLogger(Main.class.getName()).log(Level.SEVERE,
              "résultat="+resultat+ " isCancelled="+future.isCancelled());
        } catch (InterruptedException ex) {
            Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, ex);
        } catch(ExecutionException ee) {
            Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, ee);
        } catch (Exception e) {
            Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, e);
        }
    }
```

```
...
```

L'exemple ci-dessus est un extrait de code d'une application cliente Swing qui permet d'invoquer l'EJB de façon asynchrone et de tester les différents cas d'utilisation.

Le contexte de sécurité est utilisé comme lors de l'appel synchrone de la méthode.

Par contre, le contexte transactionnel n'est pas propagée à l'invocation asynchrone d'une méthode. Si la méthode invoquée est marquée avec l'attribut REQUIRES alors une nouvelle transaction est créée (comme si l'attribut REQUIRES_NEW avait été utilisé). Si la méthode invoquée est marquée avec l'attribut SUPPORT alors aucune transaction n'est utilisée. Si la méthode invoquée est marquée avec l'attribut MANDATORY alors une exception de type TransactionRequiredException est toujours levée.

53.8. L'invocation d'un EJB hors du conteneur

Dans l'exemple ci-dessous, une application standalone va invoquer un EJB déployé dans un serveur d'applications GlassFish v3.

Exemple :

```
package com.jmdoudoux.test.ebj31.client;

import com.jmdoudoux.test.ejb31.CommandeEJBRemote;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class Main {

  public static void main(String[] args) {
    Context ctx = null;
    CommandeEJBRemote bean = null;
    try {
      ctx = new InitialContext();
      bean = (CommandeEJBRemote) ctx.lookup("com.jmdoudoux.test.ejb31.CommandeEJBRemote");
      bean.valider(1234);
    } catch (NamingException ex) {
      Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, ex);
      System.exit(1);
    }
    Logger.getLogger(Main.class.getName()).log(Level.INFO, "Fin de l'application");
  }
}
```

Le nom JNDI de l'EJB est indiqué dans les logs au moment du déploiement de l'EJB dans le conteneur : il est impératif de prendre son interface Remote puisque le client ne s'exécute pas dans le contexte du serveur d'applications.

Il faut ajouter au classpath la bibliothèque qui contient l'interface de l'EJB et ajouter le fichier gf-client.jar contenu dans le sous répertoire modules du répertoire d'installation de GlassFish v3.

La bibliothèque gf-client.jar contient les valeurs des paramètres par défaut pour permettre un accès à l'annuaire via JNDI.

Si une exception de type java.net.ConnectException est levée à l'exécution, il faut préciser le port sur lequel l'application peut contacter l'annuaire via JNDI : le plus simple est de définir la propriété org.omg.CORBA.ORBInitialPort de la JVM

Résultat :

```
-Dorg.omg.CORBA.ORBInitialPort=42382
```

La valeur à utiliser est contenue dans les logs de démarrage du serveur.

54. Les services web de type Soap

Les services web de type Soap permettent l'appel d'une méthode d'un objet distant en utilisant un protocole web pour le transport (http en général) et XML pour formater les échanges. Les services web fonctionnent sur le principe du client serveur :

- un client appelle les services web
- le serveur traite la demande et renvoie le résultat au client
- le client utilise le résultat

L'appel de méthodes distantes n'est pas une nouveauté mais la grande force des services web est d'utiliser des standards ouverts et reconnus notamment HTTP et XML. L'utilisation de ces standards permet d'écrire des services web dans plusieurs langages et de les utiliser sur des systèmes d'exploitation différents.

Les services web de type Soap utilisent des messages au format XML pour permettre l'appel de méthodes ou l'échange de messages.

Certaines fonctionnalités complémentaires mais généralement utiles des services web ne sont pas encore complètement matures à cause de la jeunesse des technologies utilisées pour les mettre en oeuvre. Il reste encore de nombreux domaines à enrichir (sécurité, gestion des transactions, workflow, ...). Des technologies pour répondre à ces besoins sont en cours de développement mais généralement plusieurs solutions sont en concurrence.

Initialement, Sun a proposé un ensemble d'outils et d'API pour permettre le développement de services web avec Java. Cet ensemble se nomme JWSDP (Java Web Services Developer Pack) dont il existe plusieurs versions.

Depuis Sun a intégré la plupart de ces API permettant le développement de services web dans les spécifications de J2EE version 1.4.

La version 5 de Java EE et la version 6 de Java SE utilisent JAX-WS 2.0 pour faciliter le développement de services web en utilisant des annotations.

Ce chapitre contient plusieurs sections :

- La présentation des services web
- Les standards
- Les différents formats de services web SOAP
- Des conseils pour la mise en oeuvre
- Les API Java pour les services web
- Les implémentations des services web
- Inclure des pièces jointes dans SOAP
- WS-I
- Les autres spécifications

54.1. La présentation des services web

Les services web sont des composants distribués qui offrent des fonctionnalités aux applications au travers du réseau en utilisant des standards ouverts. Ils peuvent donc être utilisés par des applications écrites dans différents langages et exécutées dans différentes plateformes sur différents systèmes.

Les services Web utilisent une architecture distribuée composée de plusieurs ordinateurs et/ou systèmes différents qui communiquent sur le réseau. Ils mettent en oeuvre un ensemble de normes et standards ouverts qui permettent aux développeurs d'implémenter des applications distribuées internes ou externes en utilisant des outils différents fournis par différents fournisseurs.

Un service web permet généralement de proposer une ou plusieurs fonctionnalités métiers qui seront invoquées par un ou plusieurs consommateurs.

Il existe deux grandes familles de services web :

- Les services web de type SOAP
- Les services web de type REST

Ce chapitre va se concentrer sur les services web de type SOAP.

54.1.1. La définition d'un service web

Il existe plusieurs définitions pour les services web mais la plus simple pourrait être "fonctionnalité utilisable au travers du réseau en mettant en oeuvre un format standard, généralement utilisant XML".

Les services web ne sont donc qu'une nouvelle forme d'échanges de type RPC (Remote Procedure Call). Leur grand intérêt est de reposer sur des standards plutôt que sur des protocoles propriétaires. Par exemple, le transport repose généralement sur le protocol HTTP mais il est possible d'utiliser d'autres protocoles tels que JMS, FTP ou SMTP.

Les services web de type Soap font un usage intensif de XML, des namespaces XML et des schémas XML. Ces technologies font les forces des services web pour permettre leur utilisation par des clients et des serveurs hétérogènes. XML est notamment utilisé pour stocker et organiser les informations de la requête et de la réponse mais aussi pour décrire le service web. L'utilisation de XML pour le format des messages rend les échanges indépendants du système d'exploitation, de la plate-forme et du langage.

Il est ainsi possible de développer des services web avec une plate-forme (par exemple Java) et d'utiliser ces services web avec une autre plate-forme (par exemple .Net ou PHP) : c'est une des grandes forces des services web même si cela reste parfois quelque peu théorique, essentiellement à cause des implémentations des moteurs utilisés pour mettre en oeuvre les services web.

Un service web est donc une fonctionnalité accessible au travers du réseau grâce à des messages au format XML. Le format de ces messages est généralement SOAP bien que d'autres formats existent (REST, XML-RPC, ...).

Les services web peuvent prendre plusieurs formes :

- Métier
- Technique
- ...

Lors de la mise en place de services web, plusieurs problématiques interviennent tôt ou tard :

- Choix des spécifications mises en oeuvre
- Choix des outils
- Traitements proposés par les services
- Administration des services
- Orchestration des services
- ...

L'appel à un service web de type SOAP suit plusieurs étapes :

1. Le client instancie une classe de type proxy encapsulant le service Web XML.
2. Le client invoque une méthode du proxy.
3. Le moteur SOAP sur le client créé le message à partir des paramètres utilisés pour invoquer la méthode
4. Le moteur SOAP envoie le message SOAP au serveur généralement en utilisant le protocol HTTP
5. Le moteur SOAP du serveur réceptionne et analyse le message SOAP
6. Le moteur fait appel à la méthode de l'objet correspondant à la requête SOAP
7. Le moteur SOAP sur le server créé le message réponse à partir de la valeur de retour
8. Le moteur SOAP envoie le message SOAP contenant la réponse au client généralement en utilisant le protocole http
9. Le moteur SOAP du client réceptionne et analyse le message SOAP
10. Le moteur SOAP du client instancie un objet à partir du message SOAP contenant la réponse

Un des intérêts des services web est de masquer aux développeurs la complexité de l'utilisation des standards sous jacents. Ceci est réalisé grâce aux développements d'API et de moteurs pour la production et la consommation de services web.

Ces API sont dépendantes des plateformes utilisées (Java, .Net, PHP, Perl, ...) mais elles mettent toutes en oeuvre avec plus ou moins de complétude les standards de l'industrie relatifs aux services web notamment SOAP et WSDL.

Ainsi les développeurs peuvent se concentrer sur l'écriture des traitements proposés par les services et par leur consommation sans se soucier de la tuyauterie sous jacente. Un minimum de compréhension est cependant nécessaire pour bien comprendre les mécanismes mis en oeuvre.

54.1.2. Les différentes utilisations

Les services web proposent un mécanisme facilitant :

- la communication entre applications hétérogènes : un service web développé dans une technologie peut être consommé par une application développée dans une autre technologie. Ceci est possible car les services web reposent sur des standards ouverts
- l'exposition de fonctionnalités métiers aux applications internes mais aussi à des applications externes : dans ce dernier cas l'utilisation du protocole HTTP permet facilement de passer les pare-feux.
- la mise en oeuvre d'une architecture SOA puisque les services web peuvent être une implémentation technique possible d'une telle architecture

L'utilisation de services web peut avoir plusieurs intérêts :

- L'exposition de fonctionnalités au travers du réseau : les traitements des opérations des services web peuvent être invoqués via une requête HTTP, ce qui peut permettre à plusieurs applications de consommer ces services web
- La communication entre des applications et des systèmes hétérogènes : l'utilisation de standards ouverts permet la production et la consommation des services web par différentes technologies sur différents systèmes d'exploitation
- La mise en oeuvre des protocoles standards de l'industrie au niveau des couches transport, messaging, description et recherche permet de choisir entre plusieurs implémentations proposées et ainsi de ne pas dépendre d'un seul fournisseur
- Les échanges se font en utilisant l'infrastructure existante puisque les services web sont généralement invoqués en utilisant le protocole HTTP. Ceci permet de facilement passer un firewall pour permettre une invocation depuis l'extérieure
- Les services web permettent un couplage faible entre les fonctionnalités exposées et les applications qui les utilisent à tel point que les consommateurs et les producteurs peuvent être écrits pour des plateformes ou des langages différents (Java, .Net, PHP, ...).
- Les services permettent de définir de nouvelles opportunités de business voir même de nouveaux modèles économiques en permettant de proposer des fonctionnalités à des partenaires par exemple

54.2. Les standards

L'intérêt des services web grandissant, des standards ont été développés pour assurer les besoins requis pour leur mise en oeuvre.

L'architecture des services web est composée de quatre grandes couches mettant en oeuvre plusieurs technologies :

- Découverte : cette couche représente un annuaire dans lequel il est possible de publier des services et de les rechercher (UDDI est le standard)
- Description : cette couche normalise la description de l'interface publique d'un service web (WSDL (Web Service Description Language) est le standard)
- Communication : cette couche permet d'encoder les messages échangés (SOAP est le standard)
- Transport : cette couche assure le transport des messages : généralement HTTP est mis en oeuvre mais d'autres protocoles peuvent être utilisés (SMTP, FTP, ...)

La description d'un service web permet à son consommateur de connaître qu'elle est l'interface du service.

La communication permet de formaliser le format des messages échangés.

En plus de SOAP, WSDL et UDDI, il existe de nombreuses autres spécifications plus ou moins standards pour permettre la mise en oeuvre de fonctionnalités manquantes dans ces standards comme la sécurité, la gestion des transactions, l'orchestration des services, ...

Ces spécifications sont en cours de développement ou d'évolution ce qui les rend généralement immatures. De plus, fréquemment, il existe plusieurs spécifications ayant trait à un même sujet qui sont donc concurrentes. La mise en oeuvre de ces spécifications n'est pas requise pour des services web basiques mais elles peuvent être nécessaires pour des besoins plus spécifiques.

54.2.1. SOAP

SOAP (acronyme de Simple Object Acces Protocol jusqu'à sa version 1.1) est un standard du W3C qui permet l'échange formaté d'informations entre un client et un serveur. SOAP peut être utilisé pour la requête et la réponse de cet échange.

SOAP assure la partie messaging dans l'architecture des services web : il est utilisé pour normaliser le format des messages échangés entre le consommateur et le fournisseur de services web. SOAP est donc un protocole qui est principalement utilisé pour dialoguer avec des objets distribués comme peut le proposer JRMP utilisé par RMI, DCOM ou IIOP.

Son grand intérêt est d'utiliser XML ce qui le rend ouvert contrairement aux autres protocoles qui sont propriétaires : cela permet la communication entre un client et un serveur utilisant des technologies différentes. SOAP fait un usage intensif des espaces de nommages (namespaces).

SOAP est défini pour être indépendant du protocole de transport utilisé pour véhiculer le message. Cependant, le protocole le plus utilisé avec SOAP est HTTP car c'est un des protocoles le plus répandu et utilisé du fait de sa simplicité. Son utilisation avec SOAP permet rendre les services web plus interopérables. De plus, cela permet aux services web de facilement traverser les firewalls du côté producteur et consommateur notamment dans le cas d'échanges via internet.

D'autres protocoles peuvent être utilisés (par exemple SMTP ou FTP) mais leur configuration sera plus délicate car elle ne sera pas fournie en standard comme c'est le cas avec HTTP. En fait, tous les protocoles capables de véhiculer un flux d'octets peuvent être utilisés.

SOAP est aussi indépendant de tout système d'exploitation et de tout langage de programmation car il utilise XML. Ceci permet une exposition et une consommation de services web avec des outils et des OS différents.

SOAP peut être utilisé pour :

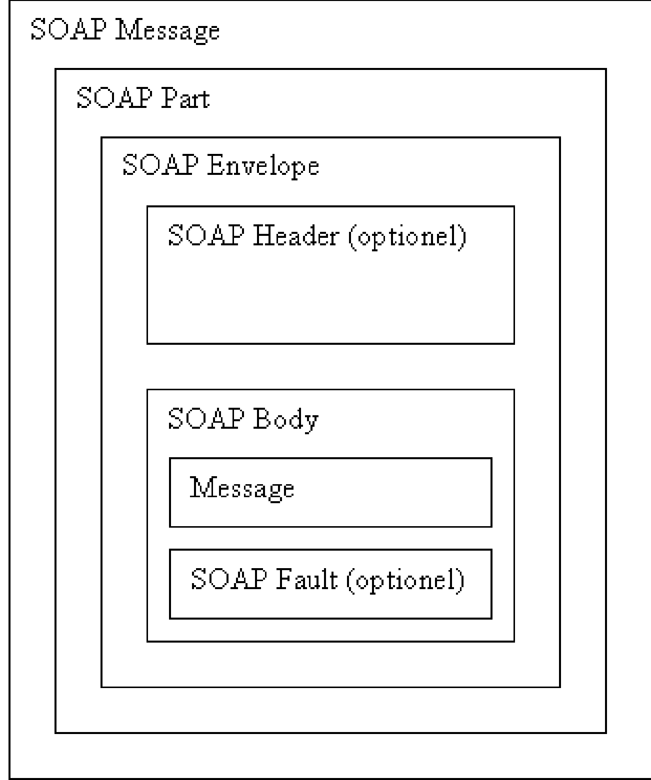
- Appeler une méthode d'un service (SOAP RPC)
- Echanger un message avec un service (SOAP Messaging)
- Recevoir un message d'un service (selon la version de Soap)

54.2.1.1. La structure des messages SOAP

Un message SOAP est contenu dans une enveloppe, ainsi le tag racine d'un document SOAP est le tag <Envelope>.

La structure d'une enveloppe SOAP se compose de plusieurs parties :

- Une entête optionnelle composée d'un ou plusieurs headers : elle contient des informations sur le traitement du message
- Un corps (Body) : il contient les informations de la requête ou de la réponse
- Une gestion d'erreurs optionnelle (Fault) contenue dans le corps
- Des pièces jointes optionnelles (attachment) contenues dans le corps



L'entête contient des informations sur le traitement du message : ces informations sont contenues dans un tag <Header>. Pour des services web simples, cette partie peut être vide ou absente mais pour des services plus complexes elle peut contenir des informations concernant les transactions, la sécurité, le routage, etc ...

Le corps du message SOAP est contenu dans un tag <Body> obligatoire. Il contient les données échangées entre le client et le service sous la forme d'un fragment de document XML.

Tous ces éléments sont codés dans le message XML avec un tag particulier mettant en oeuvre un espace de nommage particulier défini dans les spécifications de SOAP.

Un message SOAP peut aussi contenir des pièces jointes contenues chacune dans une partie optionnelle nommée AttachmentPart. Ces parties sont au même niveau que la partie SOAP Part.

SOAP définit aussi l'encodage pour les différents types de données qui est basé sur la technologie schéma XML du W3C. Les données peuvent être de type simple (chaine, entier, flottant, ...) ou de type composé.

Les types simples peuvent être

- un type de base : string, int, float, ...
- une énumération
- un tableau d'octets (array of bytes)

Les types composés

- une structure (Struct)
- un tableau (Array)

La partie SOAP Fault permet d'indiquer qu'une erreur est survenue lors des traitements du service web. Cette partie peut être composée de 4 éléments :

- faultcode : indique le type de l'erreur (VersionMismatch en cas d'incompatibilité avec la version de SOAP utilisée, MustUnderstand en cas de problème dans le header du message, Client en cas de manque d'informations de la part du client, Server en cas de problème d'exécution des traitements par le serveur)
- faultstring : message décrivant l'erreur
- faultactor : URI de l'élément ayant déclenché l'erreur
- faultdetail

54.2.1.2. L'encodage des messages SOAP

Deux formats de messages SOAP sont définis :

- remote procedure call : (RPC) permet l'invocation d'opérations qui peuvent retourner un résultat.
- message oriented (Document) : données au format XML définies dans un schéma XML

Les règles d'encodage (Encoding rules) précisent les mécanismes de sérialization des données dans un message. Il existe deux types :

- encoded : les paramètres d'entrée de la requête et les données de la réponse sont encodées en XML dans le corps du message selon un format particulier à SOAP
- literal : les données n'ont pas besoin d'être encodées : elles sont directement encodées en XML selon un schéma défini dans le WSDL

Le style et le type d'encodage permettent de définir comment les données seront sérialisées et désérialisées dans les requêtes et les réponses.

La combinaison du style et du type d'encodage peut prendre plusieurs valeurs :

- RPC/Encoded
- RPC/literal
- Document Encoded : cette combinaison n'est pas implémentée
- Document/literal
- Wrapped Document/literal : extension du Document/literal proposée par Microsoft

Le style RPC/Encoded a largement été utilisé au début des services web : actuellement ce style est en court d'abandon par l'industrie au profit du style Document/Literal. C'est pour cette raison que le style RPC/Encoded n'est pas intégré dans le WS-I Basic Profile 1.1.

Le style et le type d'encodage sont précisés dans le WSDL. L'appel du service web doit obligatoirement se faire dans le style précisé dans le WSDL puisque celui-ci détermine le format des messages échangés.

54.2.1.3. Les différentes versions de SOAP

Les versions de SOAP

- 1.0 :
- 1.1 :
- 1.2 : permet l'utilisation de requête http de type GET

Les spécifications de SOAP 1.2 sont composées de plusieurs parties :

- Part 0 : Primer
- Part 1 : Messaging Framework
- Part 2 : Adjuncts
- Specification Assertions and Test Collection

La version 1.2 des spécifications de SOAP est plus précise pour réduire les ambigüités qui pouvait conduire à des problèmes d'interopérabilité entre différentes implémentations.

SOAP 1.2 propose un support pour des protocoles de transport différents de HTTP. La sérialisation de message n'est pas obligatoirement en XML mais peut utiliser des formats binaires (XML Infoset par exemple).

54.2.2. WSDL

WSDL (acronyme de Web Service Description Language) est utilisé pour fournir une description d'un service web afin de permettre son utilisation. C'est une recommandation du W3C.

Pour permettre à un client de consommer un service web, ce dernier a besoin d'une description détaillée du service avant de pouvoir interagir avec lui. Un WSDL (Web service Description Langage) fournit cette description dans un document XML. WSDL joue un rôle important dans l'architecture des services en assurant la partie description : il contient toutes les informations nécessaires à l'invocation du service qu'il décrit.

La description WSDL d'un service web comprend une définition du service, les types de données utilisées notamment dans le cas de types complexes, les opérations utilisables, le protocole utilisé pour le transport et l'adresse d'appel.

C'est un document XML qui décrit de manière indépendante de tout langage un service web pour permettre l'appel de ses opérations et l'exploitation des réponses (les paramètres, le format des messages, le protocole utilisé, ...).

WSDL est conçu pour être indépendant de tous protocoles. Ceci rend le standard WSDL flexible mais aussi plus complexe à comprendre. Comme SOAP et HTTP sont les deux protocoles les plus couramment utilisés pour implémenter les services web, le standard WSDL intègre un support de ces deux protocoles.

L'utilisation de XML permet à des outils de différents systèmes, plateformes et langages d'utiliser le contenu d'un WSDL pour générer du code permettant de consommer un service web. Les moteurs SOAP proposent en général un outil qui va lire le WSDL et générer les classes requises pour utiliser un service web avec la technologie du moteur SOAP. Le code généré utilise un moteur SOAP qui masque toute la tuyauterie du protocole utilisé et des messages échangés lors de la consommation de services web en agissant comme un proxy. Par exemple, Axis propose l'outil WSDL2Java pour la génération de ces classes à partir du WSDL.

Pour assurer une meilleure interopérabilité, WS-I Basic Profil 1.0 oblige à utiliser WSDL et les schémas XML pour la description des services web.

Les spécifications de WSDL sont consultables à l'url : http://www.w3.org/standards/techs/wsdl#w3c_all

54.2.2.1. Le format général d'un WSDL

Un document WSDL définit plusieurs éléments :

- Type : la définition des types de données utilisés
- Message : la définition de la structure d'un message en lui attribuant un nom et en décrivant les éléments qui le composent avec un nom et un type
- PortType : la description de toutes les opérations proposées par le service web (interface du service) et identification de cet ensemble avec un nom
- Operation : la description d'une action proposée par le service web notamment en précisant les messages en entrée (input) et en sortie (output)
- Binding : la description du protocole de transport et d'encodage utilisé par un PortType afin de pouvoir invoquer un service web
- Port : référence un Binding (généralement cela correspond à l'url d'invocation du service web)
- Service : c'est un ensemble de ports

Un WSDL est un document XML dont le tag racine est <definitions> et qui utilise l'espace de nommage "http://schemas.xmlsoap.org/wsdl/".

Un WSDL est virtuellement composé de deux parties :

- des définitions abstraites : celles-ci concernent l'interface du service (types, message, portType). Ces informations sont exploitées dans le code du client
- des définitions concrètes : celles-ci concernent l'invocation du service (binding, service). Ces informations sont exploitées par le moteur SOAP.

Le contenu du WSDL d'un service nommé MonService est de la forme :

Exemple :

```
<!-Structure d'un WSDL -->
<definitions name="MonService"
targetNamespace="http://com.jmdoudoux.test.ws.monservice/"
xmlns="http://schemas.xmlsoap.org/wsdl/">
   <!-- Définitions abstraites -->
   <types> ... </types>
   <message> ... </message>
   <portType> ... </portType>

   <!-- Définitions concrètes -->
   <binding> ... </binding>
   <service> ... </service>
</definition>
```

L'ordre de définition des informations dans un WSDL facilite les traitements par une machine de ce document. Pour une exploitation par un humain, il est plus facile de lire le WSDL à l'envers (en commençant par la fin).

Exemple :

```
<?xml version="1.0" encoding="UTF-8" ?>
 <wsdl:definitions targetNamespace="http://axis.test.jmdoudoux.com"
      xmlns:apachesoap="http://xml.apache.org/xml-soap"
      xmlns:impl="http://axis.test.jmdoudoux.com"
      xmlns:intf="http://axis.test.jmdoudoux.com"
      xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
      xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
      xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!--
WSDL created by Apache Axis version: 1.3
Built on Oct 05, 2005 (05:23:37 EDT)

  -->
  <wsdl:message name="additionnerRequest">
  <wsdl:part name="valeur1" type="xsd:int" />
  <wsdl:part name="valeur2" type="xsd:int" />
  </wsdl:message>
  <wsdl:message name="additionnerResponse">
  <wsdl:part name="additionnerReturn" type="xsd:long" />
  </wsdl:message>
 <wsdl:portType name="Calculer">
  <wsdl:operation name="additionner" parameterOrder="valeur1 valeur2">
  <wsdl:input message="impl:additionnerRequest" name="additionnerRequest" />
  <wsdl:output message="impl:additionnerResponse" name="additionnerResponse" />
  </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="CalculerSoapBinding" type="impl:Calculer">
  <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http" />
  <wsdl:operation name="additionner">
  <wsdlsoap:operation soapAction="" />
  <wsdl:input name="additionnerRequest">
  <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    namespace="http://axis.test.jmdoudoux.com" use="encoded" />
  </wsdl:input>
  <wsdl:output name="additionnerResponse">
```

```
<wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
  namespace="http://axis.test.jmdoudoux.com" use="encoded" />
</wsdl:output>
</wsdl:operation>
</wsdl:binding>
<wsdl:service name="CalculerService">
<wsdl:port binding="impl:CalculerSoapBinding" name="Calculer">
<wsdlsoap:address location="http://localhost:8080/TestWS/services/Calculer" />
</wsdl:port>
</wsdl:service>
</wsdl:definitions>
```

Un document WSDL est un document XML dont le tag racine est <definitions>. Généralement, ce tag contient la définition des différents espaces de nommage qui seront utilisés dans le document XML.

L'espace de nommage du document WSDL est défini.

Exemple :

```
xmlns="http://schemas.xmlsoap.org/wsdl/"  (déclaration de l'espace de nommage par défaut)
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
```

L'espace de nommage de la norme Schema XML est défini.

Exemple :

```
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
```

Plusieurs autres espaces de nommages standards sont généralement définis selon les protocoles utilisés.

Exemple :

```
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
```

Il est possible de trouver la définition d'espaces de nommage propre à l'implémentation.

Exemple :

```
xmlns:apachesoap="http://xml.apache.org/xml-soap"
```

Un ou plusieurs espaces de nommage sont définis pour le service web lui-même.

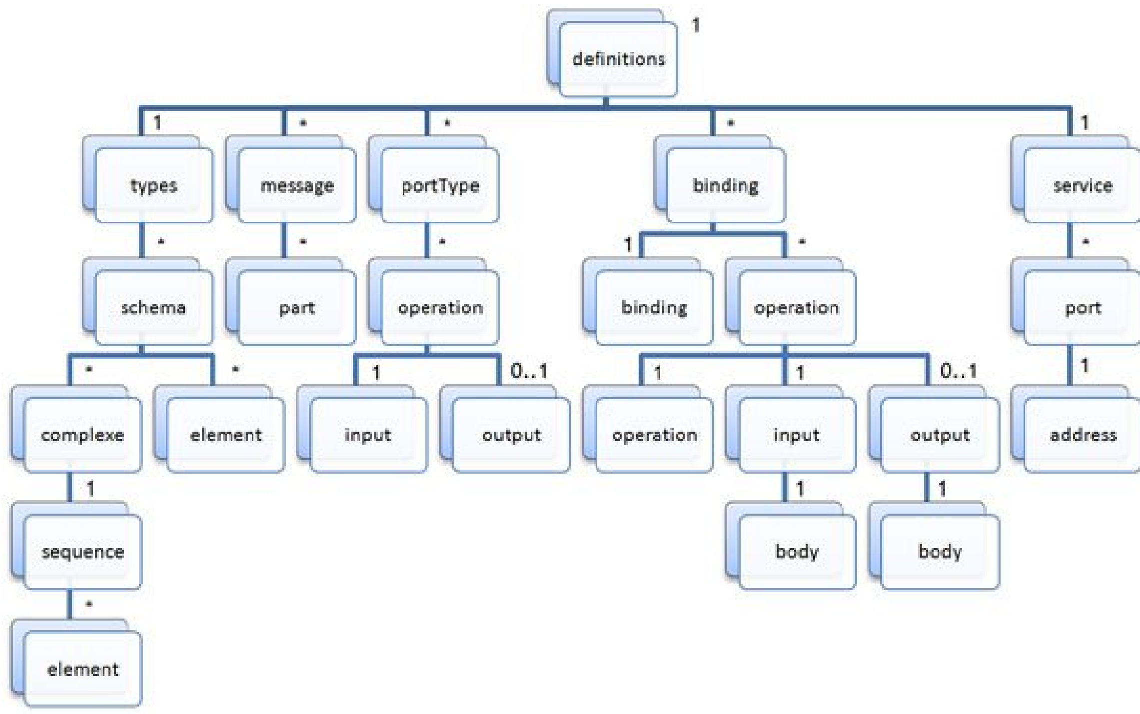
Remarque : les préfixes utilisés pour ces espaces de nommages peuvent être différents selon l'implémentation des services web mise en oeuvre

Dans un document WSDL, les différentes entités font référence entre elles grâce à leur nom complet (espace de nommage et nom).

L'élément racine d'un WDSL est le tag <definitions>.

Le tag <definitions> peut contenir plusieurs tags fils :

- <types> : description des types de données utilisés
- <message> : description des messages qui peuvent être composés de plusieurs types
- <portType> : description des opérations du endpoint sous la forme d'échange de messages. Ceci correspond à l'interface du service
- <binding> : description du protocole et spécification du format des données pour un portType
- <service> : description des endpoints du service (binding et uri)



54.2.2.2. L'élément Types

Le tag <definitions> ne peut avoir qu'un seul tag fils <types>.

L'élément <types> contient une définition des différents types de données qui seront utilisés.

Cette définition peut être faite sous plusieurs formats mais l'utilisation des schémas XML est recommandée. Le WS-I Basic Profile impose que cette description soit faite avec des schémas XML.

L'élément <types> peut avoir aucun, un ou plusieurs éléments fils <schema> ayant pour espace de nommage "http://www.w3.org/2001/XMLSchema".

Chaque structure de données est décrite en utilisant la norme schéma XML.

L'élément <types> peut donc avoir plusieurs schémas XML comme éléments fils. Ces schémas peuvent décrire des types simples (element) ou complexes (complexType).

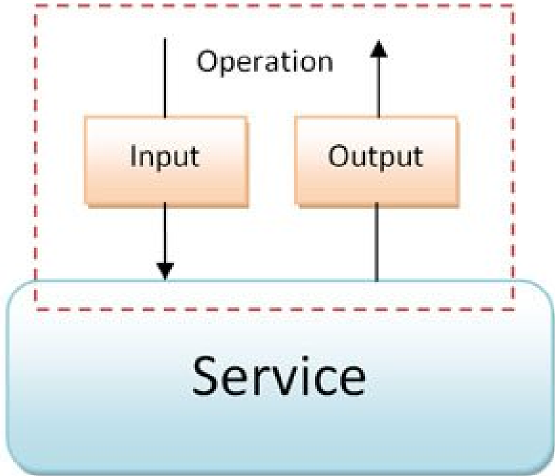
54.2.2.3. L'élément Message

Le tag <definitions> peut avoir plusieurs tags fils <message>. Le tag <message> décrit un message qui est utilisé en tant que requête ou réponse lors de l'invocation d'une opération : il contient une définition des paramètres pour un message échangé en entrée ou en sortie..

Le tag <message> peut avoir un ou plusieurs tags <part>. Le tag <part> possède un attribut "name" pour permettre d'y faire référence et utilise un attribut "element" (pour le style document qui représente l'élément XML inséré dans le body) ou un attribut "type" (pour le style RPC qui représente les paramètres de l'opération).

54.2.2.4. L'élément PortType/Interface

En WSDL, un échange de messages est une opération qui peut donc avoir une requête en entrée et une réponse en sortie.



Le tag <definitions> peut avoir un ou plusieurs tags fils <typePort>. Le tag <portType> décrit l'interface d'un service web.

Le terme typePort est particulièrement ambigu : il correspond à une description de l'interface du service. Le tag <interface> est utilisé à partir de la version 2.0 de WSDL.

Le tag <typePort> possède un attribut name qui permet d'y faire référence.

Il contient un ensemble d'opérations chacune définie dans un tag fils <operation> qui possède un attribut name qui permet d'y faire référence.

Le tag <operation> peut avoir les tags fils <input>, <output> et <fault>. La présence et l'ordre de deux premiers tags définissent le mode d'invocation d'une opération, nommé MEP (Message Exchance Pattern)

MEP	Description
One-way	L'endpoint reçoit un message sans fournir de réponse : <input> uniquement
Request-response	L'endpoint reçoit un message et envoie la réponse correspondante : <input> et <output>
Notification	L'endpoint envoie un message sans avoir de réponse : <output> uniquement
Solicit-response	L'endpoint envoie un message et reçoit la réponse correspondante : <output> et <input>

Le support de ces types d'opérations dépend de l'implémentation du moteur SOAP utilisé.

L'attribut message des tags <input> et <output> font référence à un tag <message> via son nom.

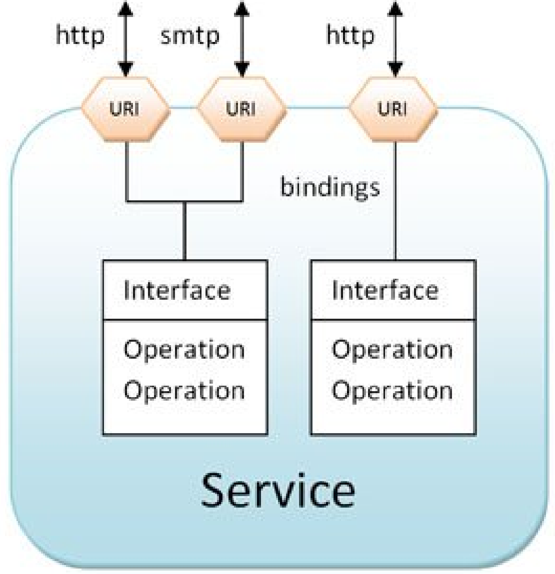
54.2.2.5. L'élément Binding

La description du service doit aussi fournir des informations pour invoquer le service :

- le protocole utilisé pour le transport du message
- l'encodage du message : style et mécanisme d'encodage

Un binding permet de fournir des détails sur la façon dont les données sont transportées.

Un service peut avoir plusieurs bindings mais chacun doit se faire sur une URI unique nommé endpoint.



Le tag <definitions> peut avoir un ou plusieurs tags fils <binding>.

Il permet de définir pour un portType le protocole de transport utilisé et le mode d'encodage des messages.

Le tag <binding> possède un attribut name qui permet d'y faire référence et un attribut type qui permet de faire référence au portType concerné via son nom.

Le détail des informations sur le protocole et le mode d'encodage sont des extensions spécifiques notamment une fournie en standard relative à SOAP.

Ainsi le tag fils <soap:binding> est utilisé pour préciser que c'est la version 1.1 de SOAP qui sera utilisée. Son attribut style permet de préciser le style du message : les valeurs possibles sont rpc ou document. Son attribut transport permet de préciser le procotole de transport à utiliser, généralement "http://schemas.xmlsoap.org/soap/http" pour le protocole http.

Le tag <binding> possède un tag fils <operation> pour chaque opération. Le tag <operation> peut avoir plusieurs tags fils.

Le tag fils <soap:operation> permet de définir via son attribut "soapAction" la valeur du header http correspondant. Selon les spécifications WS-I BP, l'attribut "soapAction" doit toujours avoir la valeur chaîne vide.

Les tags <input> et <output> permettent de fournir des précisions sur l'encodage du corps des messages.

Le tag fils <soap:body> permet de préciser comment le corps de message sera encodé via son attribut "use" : les valeurs possibles sont encoded et document.

L'utilisation de l'encodage "rpc" précise que le corps du message sera une représentation XML des paramètres ou de la valeur de retour de la méthode invoquée.

L'utilisation de l'encodage "document" précise que le corps du message sera un message XML.

54.2.2.6. L'élément Service

Le tag <definitions> ne peut avoir qu'un seul tag fils <service>.

Un service possède un nom précisé dans la valeur de son attribut name.

Un service est composé d'un ou plusieurs ports qui en SOAP correspondent à des endpoints. Chaque port est associé à un binding en utilisant l'attribut binding qui a comme valeur le nom d'un binding défini.

Les tags fils du tag <port> sont spécifiques au binding utilisé : ce sont des extensions spécifiques qui précisent le endpoint selon le binding. Par exemple pour préciser l'url d'un service web utilisant http, il faut utiliser le tag <address> en fournissant l'url du endpoint comme valeur à l'attribut location. Le tag <endpoint> est utilisé à partir de la version 2.0 de WSDL.

Un port permet de décrire la façon d'accéder au service, ce qui correspond généralement à l'url d'un endpoint et à un binding.

54.2.3. Les registres et les services de recherche

54.2.3.1. UDDI

UDDI, acronyme de Universal Description, Discovery and Integration, est utilisé pour publier et rechercher des services web. C'est un protocole et un ensemble de services pour utiliser un annuaire afin de stocker les informations concernant les services web et de permettre à un client de les retrouver. Les spécifications sont rédigées par l'Oasis.

UDDI est une spécification pour permettre la publication et la recherche d'informations sur des entreprises et les services web qu'elles proposent. UDDI permet à une entreprise de s'inscrire dans l'annuaire et d'enregistrer et de publier ses services web. Il est alors possible d'accéder à l'annuaire et de rechercher un service web.

Le site web officiel d'UDDI est à l'url : http://uddi.xml.org

Un annuaire UDDI contient une description de services web mais aussi des entreprises qui les proposent. Ainsi, il est possible avec UDDI de faire une recherche par entreprise ou par activité.

Les données incluses dans un annuaire UDDI sont classifiées dans trois catégories :

- les pages blanches (white pages) : elles contiennent les informations générales sur une entreprise
- les pages jaunes (yellow pages) : elles permettent une catégorisation des entreprises
- les pages vertes (green pages) : elles contiennent les informations techniques sur les services proposés

Il est possible d'utiliser un annuaire UDDI en interne dans une entreprise mais il existe aussi des annuaires UDDI globaux nommés UBR (UDDI Business Registry).

Il existe plusieurs versions d'UDDI :

- version 1 :
- version 2 :
- version 3 : les spécifications de cette version sont consultables à l'url http://uddi.org/pubs/uddi_v3.htm

UDDI n'est pas un élément indispensable à la mise en oeuvre des services web comme peut l'être XML, WSDL ou SOAP.

UDDI est une spécification pour un annuaire dont l'accès aux fonctionnalités se fait sous la forme de services web de type SOAP pour les recherches et les mises à jour.

Le site http://xmethods.net propose une liste des services web publics.

54.2.3.2. Ebxml

La suite de cette section sera développée dans une version future de ce document

54.3. Les différents formats de services web SOAP

Un message SOAP peut être formaté de plusieurs façons en fonction de son style et de son type d'encodage.

Il existe deux styles de services web reposant sur SOAP : RCP et Document.

En plus du style, il existe deux types d'encodage : Encoded et Literal. Cela permet de définir quatre combinaisons mais généralement les combinaisons utilisées sont RPC/Encoded et Document/Literal. La combinaison Document/Encoded n'est supportée par aucune implémentation.

De plus, Microsoft est à l'origine d'un cinquième format qui bien que non standardisé est largement utilisé car il est mis en oeuvre par défaut dans la plate-forme.Net et il offre un bon compromis entre performance et restrictions d'utilisation.

Style / Type d'encodage	Encoded	Literal
RPC	RPC / Encoded	RPC / Literal
Document	Document / Encoded	Document / Literal
		Document / Literal wrapped

Il y a deux façons de structurer un document SOAP : RPC et Document. Initialement, avant la diffusion de sa première version, SOAP ne permettait que le style RPC. La version 1.0 s'est vue ajouter le support pour le style Document.

Le style RPC est parfaitement structuré alors que le type Document n'a pas de structure imposée mais son contenu peut être facilement validé grâce à un schéma XML ou traité puisque c'est un document XML. Avec le style document, il est donc possible de structurer librement le corps du message grâce au schéma XML.

Les différents styles sont :

Style	Description
RPC	Les messages contiennent le nom de l'opération Paramètres en entrée multiple et valeur de retour
Document	Les messages ne contiennent pas le nom de l'opération Un document XML en entrée et en retour

Les deux désignations pour le style d'encodage (RPC et document) peuvent être trompeuses car elles peuvent induire que le style RPC est utilisé pour l'invocation d'opérations distantes et que le style document est utilisé pour l'échange de messages. En fait, le style n'a rien à voir avec un modèle de programmation mais il permet de préciser comment le message SOAP est encodé.

Dans le style RPC, le corps du message (tag <soap:body>) contient un élément qui est le nom de l'opération du service. Cet élément contient un élément fils pour chaque paramètre.

Dans le style document, le corps du message (tag <soap:body>) contient directement un document xml dont toutes les composants doivent être décrit dans un ou plusieurs schémas XML. Le moteur Soap est alors responsable du mapping entre le contenu du message et les objets du serveur

Les types d'encodage pour sérialiser les messages en XML sont :

Encodage	Description
Encoded	Aussi appelé SOAP encoding car l'encodage est spécifique à SOAP sans utiliser de schéma XML
Literal	L'encodage du message repose sur les schémas XML
Literal wrapped	Idem Literal avec en plus l'encapsulation de chaque message dans un tag qui contient le nom de l'opération. Ce format est défini par Microsoft qui l'utilise dans sa plate-forme .Net

Le type d'encodage Literal propose que le contenu du corps du body soit validé par un schéma XML donné : chaque élément qui correspond à un paramètre ou à la valeur de retour est décrit dans un schéma XML.

Le type d'encodage Encoded utilise un ensemble de règles reposant sur les types de données des schémas XML pour encoder les données mais le message ne respecte pas de schéma particulier : chaque élément qui correspond à un paramètre ou à la valeur de retour contient la description de la donnée sous la forme d'attributs spécifiés dans la norme Soap (type, null ou pas, ...)

Le type d'encodage Encoded est particulièrement adapté lors de l'utilisation d'un graphe d'objets cyclique car chaque type d'objet ne sera défini qu'une seule fois. Avec l'encodage Literal, chaque élément est répété dans le document.

Le type d'encodage Literal permet une manipulation du document XML qui constitue le message (validation par un schéma XML, parsing du document, transformation à l'aide d'une feuille de style XSLT, ...)

Le format RPC encoded repose sur des types définis par SOAP alors que les formats RPC literal et Document Literal repose sur les types du schéma XML.

Il existe donc plusieurs formats utilisables pour un message SOAP :

- RPC Encoded : c'est le premier format historiquement proposé par SOAP mais son utilisation est de moins en moins fréquente
- RPC Literal :
- Document Literal :
- Document Literal Wrapped :
- Document Encoded : ce format n'est pas supporté actuellement par les moteurs de services web

Ces différents styles et types d'encodage sont à l'origine de difficultés d'interopérabilité au début des services web car la plate-forme.Net utilisait le style Document par défaut et les implémentations sur la plate-forme Java utilisaient plutôt le style RPC.

Au début de l'utilisation de SOAP, c'est donc le format RPC encoding qui était utilisé. Depuis, c'est plutôt le format Document/literal ou RPC/Literal qui est recommandé notamment par les spécifications WS-I Basic Profile.

Les sections suivantes vont utiliser la classe d'implémentation du service web ci-dessous avec Axis

Exemple :

```
package com.jmdoudoux.test.axis;

public class Calculer {

        public long additionner(int valeur1, int valeur2) {
               return valeur1+valeur2;
        }

}
```

Les sections suivantes vont utiliser la classe d'implémentation du service web ci-dessous avec Metro

Exemple :

```
package com.jmdoudoux.test.ws;
```

```
import javax.jws.WebService;

@WebService
public class Calculer {

        public long additionner(int valeur1, int valeur2) {
                return valeur1+valeur2;
        }

}
```

54.3.1. Le format RPC Encoding

Ce format de messages est le plus ancien et le plus simple à mettre en oeuvre pour le développeur.

La structure du corps du message avec le style RPC est imposée. Par exemple, pour une requête, il doit contenir le nom de la méthode ainsi que ses paramètres. Cette structure est donc de la forme :

Exemple :

```
<soapenv:Body>
<ns0:nom_de_la_methode xmlns:ns0="uri_de_l_espace_de_nommage"
  soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <nom_param_1 xsi:type="xsd:type_param_1">valeur_parametre_1</nom_param_1>
  <nom_param_2 xsi:type="xsd:type_param_2">valeur_parametre_2</nom_param_2>
  </ns0: nom_de_la_methode>
  </soapenv:Body>
  </soapenv:Envelope>
```

Le message réponse a une forme similaire.

Le type des paramètres peut être simple ou plus complexe (par exemple un objet qui encapsule des données de types simple ou d'autres objets).

Attention : RPC Encoding n'est pas non-conforme à la spécification WS-I Basic Profile

Exemple : Le fichier WSDL

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://axis.test.jmdoudoux.com"
                  xmlns:apachesoap="http://xml.apache.org/xml-soap"
                  xmlns:impl="http://axis.test.jmdoudoux.com"
                  xmlns:intf="http://axis.test.jmdoudoux.com"
                  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
                  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<!--WSDL created by Apache Axis version: 1.3
Built on Oct 05, 2005 (05:23:37 EDT)-->
   <wsdl:message name="additionnerRequest">
      <wsdl:part name="valeur1" type="xsd:int"/>
      <wsdl:part name="valeur2" type="xsd:int"/>
   </wsdl:message>
   <wsdl:message name="additionnerResponse">
      <wsdl:part name="additionnerReturn" type="xsd:long"/>
   </wsdl:message>
   <wsdl:portType name="Calculer">
      <wsdl:operation name="additionner" parameterOrder="valeur1 valeur2">
         <wsdl:input message="impl:additionnerRequest" name="additionnerRequest"/>
         <wsdl:output message="impl:additionnerResponse" name="additionnerResponse"/>
      </wsdl:operation>
   </wsdl:portType>
```

```
   <wsdl:binding name="CalculerSoapBinding" type="impl:Calculer">
      <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
      <wsdl:operation name="additionner">
         <wsdlsoap:operation soapAction=""/>
         <wsdl:input name="additionnerRequest">
            <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
              namespace="http://axis.test.jmdoudoux.com" use="encoded"/>
         </wsdl:input>
         <wsdl:output name="additionnerResponse">
            <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
              namespace="http://axis.test.jmdoudoux.com" use="encoded"/>
         </wsdl:output>
      </wsdl:operation>
   </wsdl:binding>
   <wsdl:service name="CalculerService">
      <wsdl:port binding="impl:CalculerSoapBinding" name="Calculer">
         <wsdlsoap:address location="http://localhost:8080/TestWS/services/Calculer"/>
      </wsdl:port>
   </wsdl:service>
</wsdl:definitions>
```

Exemple : descripteur Axis

Exemple :

```
<service name="Calculer" provider="java:RPC">
  <operation name="additionner" qname="ns1:additionner"
             returnQName="additionnerReturn"
             returnType="xsd:long"
             soapAction=""
             xmlns:ns1="http://axis.test.jmdoudoux.com"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <parameter name="valeur1" type="xsd:int"/>
   <parameter name="valeur2" type="xsd:int"/>
  </operation>
  <parameter name="allowedMethods" value="additionner"/>
  <parameter name="typeMappingVersion" value="1.2"/>
  <parameter name="wsdlPortType" value="Calculer"/>
  <parameter name="className" value="com.jmdoudoux.test.axis.Calculer"/>
  <parameter name="wsdlServicePort" value="Calculer"/>
  <parameter name="wsdlTargetNamespace" value="http://axis.test.jmdoudoux.com"/>
  <parameter name="wsdlServiceElement" value="CalculerService"/>
 </service>
```

La requête SOAP

Exemple :

```
<?xml version="1.0" encoding="UTF-8" ?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <ns0:additionner
      xmlns:ns0="http://axis.test.jmdoudoux.com"
      soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      <valeur1 xsi:type="xsd:int">10</valeur1>
      <valeur2 xsi:type="xsd:int">20</valeur2>
    </ns0:additionner>
  </soapenv:Body>
</soapenv:Envelope>
```

La réponse SOAP

Exemple :

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <ns1:additionnerResponse
      xmlns:ns1="http://axis.test.jmdoudoux.com"
      soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      <additionnerReturn href="#id0" />
    </ns1:additionnerResponse>
    <multiRef xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
              id="id0" soapenc:root="0"
              soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
              xsi:type="xsd:long">30</multiRef>
  </soapenv:Body>
</soapenv:Envelope>
```

Avantages

- Ce format est facilement compréhensible par un être humain.
- le nom de l'opération à invoquer est inclus dans le message ce qui rend le dispatching par le moteur SOAP vers la méthode correspondant très facile
- la gestion des valeurs null est supportée en standard

Inconvénients

- Il n'est pas possible de valider le message dans la mesure ou seuls les paramètres sont définis dans un schéma. Le reste du contenu du corps de l'enveloppe est défini directement dans le WSDL.
- Ce mode peut poser des problèmes d'intéropérabilité : il est d'ailleurs incompatible avec les spécifications WS-I Basic Profile. Son utilisation n'est donc plus recommandée
- C'est le type d'encodage qui a les moins bonnes performances et qui génère les messages les plus verbeux notamment à cause de la présence pour chaque donnée de son type

54.3.2. Le format RPC Literal

Les messages de type RPC/Literal sont encodés comme des appels RPC avec une description des paramètres et des valeurs de retour décrites chacunes avec son propre schéma XML.

Le moteur Soap utilisé pour l'exemple de cette section est Metro version 1.5.

Exemple :

```
import javax.jws.soap.SOAPBinding.Use;

@WebService
@SOAPBinding(style=Style.RPC, use=Use.LITERAL, parameterStyle=ParameterStyle.BARE)
public class Calculer {

  public long additionner(Valeurs valeurs) {
    return valeurs.getValeur1()+valeurs.getValeur2();
  }
}
```

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<endpoints xmlns="http://java.sun.com/xml/ns/jax-ws/ri/runtime"
      version="2.0">
  <endpoint name="CalculerWS" implementation="com.jmdoudoux.test.ws.Calculer"
      url-pattern="/services/CalculerWS" />
</endpoints>
```

Le fichier WSDL

Exemple :

```
<?xml version='1.0' encoding='UTF-8'?>
  <!--
    Published by JAX-WS RI at http://jax-ws.dev.java.net. RI's version is
    JAX-WS RI 2.1.7-hudson-48-.
  -->
  <!--
    Generated by JAX-WS RI at http://jax-ws.dev.java.net. RI's version is
    JAX-WS RI 2.1.7-hudson-48-.
  -->
<definitions
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:tns="http://ws.test.jmdoudoux.com/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns="http://schemas.xmlsoap.org/wsdl/"
  targetNamespace="http://ws.test.jmdoudoux.com/" name="CalculerService">
  <types>
    <xsd:schema>
      <xsd:import namespace="http://ws.test.jmdoudoux.com/"
        schemaLocation="http://localhost:8089/TestMetro/services/CalculerWS?xsd=1" />
    </xsd:schema>
  </types>
  <message name="additionner">
    <part name="additionner" element="tns:additionner" />
  </message>
  <message name="additionnerResponse">
    <part name="additionnerResponse" element="tns:additionnerResponse" />
  </message>
  <portType name="Calculer">
    <operation name="additionner">
      <input message="tns:additionner" />
      <output message="tns:additionnerResponse" />
    </operation>
  </portType>
  <binding name="CalculerPortBinding" type="tns:Calculer">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
      style="rpc" />
    <operation name="additionner">
      <soap:operation soapAction="" />
      <input>
        <soap:body use="literal" namespace="http://ws.test.jmdoudoux.com/" />
      </input>
      <output>
        <soap:body use="literal" namespace="http://ws.test.jmdoudoux.com/" />
      </output>
    </operation>
  </binding>
  <service name="CalculerService">
    <port name="CalculerPort" binding="tns:CalculerPortBinding">
      <soap:address location="http://localhost:8089/TestMetro/services/CalculerWS" />
    </port>
  </service>
</definitions>
```

La description est faite dans un schéma dédié

Exemple :

```
<?xml version='1.0' encoding='UTF-8'?>
<xs:schema xmlns:tns="http://ws.test.jmdoudoux.com/"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           version="1.0" targetNamespace="http://ws.test.jmdoudoux.com/">
<xs:element name="additionner" nillable="true" type="tns:valeurs" />
  <xs:element name="additionnerResponse" type="xs:long" />
  <xs:complexType name="valeurs">
    <xs:sequence>
      <xs:element name="valeur1" type="xs:int" />
      <xs:element name="valeur2" type="xs:int" />
    </xs:sequence>
</xs:complexType>
```

```
</xs:schema>
```

Le premier élément du tag <body> du message désigne la méthode à invoquer

La requête SOAP

Exemple :

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:ws="http://ws.test.jmdoudoux.com/">
   <soapenv:Header/>
   <soapenv:Body>
      <ws:additionner>
         <ws:additionner>
            <valeur1>20</valeur1>
            <valeur2>30</valeur2>
         </ws:additionner>
      </ws:additionner>
   </soapenv:Body>
</soapenv:Envelope>
```

La réponse SOAP

Exemple :

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Body>
      <ns2:additionnerResponse xmlns:ns2="http://ws.test.jmdoudoux.com/">
      0</ns2:additionnerResponse>
   </S:Body>
</S:Envelope>
```

Avantages

- Supporté par le WS-I Basic Profile
- Le nom de la méthode invoquée est inclus dans le message

Inconvénients

54.3.3. Le format Document encoding

Ce type d'encodage n'est supporté par aucun moteur de services web.

54.3.4. Le format Document literal

Pour respecter le WS-I BP, le tag <soap:body> d'un message Soap encodé en document literal ne peut avoir qu'un seul élément fils.

L'exemple de cette section va donc utiliser une version légèrement différente du service web qui attend en paramètre un bean qui encapsule les données à calculer.

Exemple :

```
package com.jmdoudoux.test.ws.axis;
public class CalculerWS {

            public
```

```
long additionner(Valeurs valeurs) {
                        return
valeurs.getValeur1()+valeurs.getValeur2();
            }
}
```

La classe Valeurs est un POJO qui encapsule les paramètres requis par la méthode.

Exemple :

```
package com.jmdoudoux.test.ws;

public class Valeurs {
  private int valeur1;
  private int valeur2;

  public Valeurs() {
    super();
  }

  public Valeurs(int valeur1, int valeur2) {
    super();
    this.valeur1 = valeur1;
    this.valeur2 = valeur2;
  }

  public synchronized int getValeur1() {
    return valeur1;
  }

  public synchronized void setValeur1(int valeur1) {
    this.valeur1 = valeur1;
  }

  public synchronized int getValeur2() {
    return valeur2;
  }

  public synchronized void setValeur2(int valeur2) {
    this.valeur2 = valeur2;
  }

}
```

Descripteur Axis 1.x

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
            xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
  <service name="CalculerWS" provider="java:RPC" style="document"
    use="literal">
    <parameter name="wsdlTargetNamespace" value="http://axis.ws.test.jmdoudoux.com" />
    <parameter name="wsdlServiceElement" value="CalculerWSService" />
    <parameter name="schemaQualified" value="http://axis.ws.test.jmdoudoux.com" />
    <parameter name="wsdlServicePort" value="CalculerWS" />
    <parameter name="className" value="com.jmdoudoux.test.ws.axis.CalculerWS" />
    <parameter name="wsdlPortType" value="CalculerWS" />
    <parameter name="typeMappingVersion" value="1.2" />
    <operation xmlns:retNS="http://axis.ws.test.jmdoudoux.com"
      xmlns:rtns="http://www.w3.org/2001/XMLSchema" name="additionner"
      qname="additionner" returnQName="retNS:additionnerReturn" returnType="rtns:long"
      soapAction="">
      <parameter xmlns:pns="http://axis.ws.test.jmdoudoux.com"
        xmlns:tns="http://axis.ws.test.jmdoudoux.com" qname="pns:valeurs"
        type="tns:Valeurs" />
    </operation>
    <parameter name="allowedMethods" value="additionner" />
```

```
    <typeMapping xmlns:ns="http://axis.ws.test.jmdoudoux.com"
      qname="ns:Valeurs" type="java:com.jmdoudoux.test.ws.axis.Valeurs"
      serializer="org.apache.axis.encoding.ser.BeanSerializerFactory"
      deserializer="org.apache.axis.encoding.ser.BeanDeserializerFactory"
      encodingStyle="" />
  </service>
</deployment>
```

Le fichier WSDL

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://axis.ws.test.jmdoudoux.com"
                  xmlns:apachesoap="http://xml.apache.org/xml-soap"
                  xmlns:impl="http://axis.ws.test.jmdoudoux.com"
                  xmlns:intf="http://axis.ws.test.jmdoudoux.com"
                  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<!--WSDL created by Apache Axis version: 1.4
Built on Apr 22, 2006 (06:55:48 PDT)-->
 <wsdl:types>
  <schema elementFormDefault="qualified"
          targetNamespace="http://axis.ws.test.jmdoudoux.com"
          xmlns="http://www.w3.org/2001/XMLSchema">
   <complexType name="Valeurs">
    <sequence>
     <element name="valeur1" type="xsd:int"/>
     <element name="valeur2" type="xsd:int"/>
    </sequence>
   </complexType>
   <element name="valeurs" type="impl:Valeurs"/>
   <element name="additionnerReturn" type="xsd:long"/>
  </schema>
 </wsdl:types>
   <wsdl:message name="additionnerResponse">
      <wsdl:part element="impl:additionnerReturn" name="additionnerReturn"/>
   </wsdl:message>
   <wsdl:message name="additionnerRequest">
      <wsdl:part element="impl:valeurs" name="valeurs"/>
   </wsdl:message>
   <wsdl:portType name="CalculerWS">
      <wsdl:operation name="additionner" parameterOrder="valeurs">
         <wsdl:input message="impl:additionnerRequest" name="additionnerRequest"/>
         <wsdl:output message="impl:additionnerResponse" name="additionnerResponse"/>
      </wsdl:operation>
   </wsdl:portType>
   <wsdl:binding name="CalculerWSSoapBinding" type="impl:CalculerWS">
      <wsdlsoap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
      <wsdl:operation name="additionner">
         <wsdlsoap:operation soapAction=""/>
         <wsdl:input name="additionnerRequest">
            <wsdlsoap:body use="literal"/>
         </wsdl:input>
         <wsdl:output name="additionnerResponse">
            <wsdlsoap:body use="literal"/>
         </wsdl:output>
      </wsdl:operation>
   </wsdl:binding>
   <wsdl:service name="CalculerWSService">
      <wsdl:port binding="impl:CalculerWSSoapBinding" name="CalculerWS">
         <wsdlsoap:address location="http://localhost:8089/TestsAxisWS/services/CalculerWS"/>
      </wsdl:port>
   </wsdl:service>
</wsdl:definitions>
```

La requête SOAP

Exemple :

```
<soapenv:Envelope
   xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:axis="http://axis.ws.test.jmdoudoux.com">
   <soapenv:Header/>
   <soapenv:Body>
      <axis:valeurs>
         <axis:valeur1>20</axis:valeur1>
         <axis:valeur2>30</axis:valeur2>
      </axis:valeurs>
   </soapenv:Body>
</soapenv:Envelope>
```

La réponse SOAP

Exemple :

```
<soapenv:Envelope
   xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <soapenv:Body>
      <additionnerReturn
        xmlns="http://axis.ws.test.jmdoudoux.com">
        50
      </additionnerReturn>
   </soapenv:Body>
</soapenv:Envelope>
```

Avantages

- Le contenu du corps de l'enveloppe peut être validé puisque que tous les éléments sont définis dans un schéma
- Supporté par le WS-I Basic Profile avec quelques contraintes

Inconvénients

- Le fichier WSDL est plus compliqué à comprendre pour un être humain.
- Le nom de l'opération n'apparaît pas dans la requête SOAP : le mapping vers la méthode du service web à invoquer est donc plus limité puisqu'il doit se faire sur la séquence de paramètres
- Il n'est donc pas possible dans un même service web d'avoir deux méthodes avec la même liste de paramètres
- Le nom de l'opération n'est pas contenu dans le message ce qui impose des contraintes au niveau des signatures des interfaces des opérations
- pour respecter le WS-I BP le tag <soap:body> ne peut avoir qu'un seul tag fils

54.3.5. Le format Document Literal wrapped

Ce format a été défini par Microsoft pour la plate-forme.Net et il n'existe aucune spécification officielle mais c'est un standard de fait.

Ce format reprend le format Document Literal mais le corps contient un élément qui précise le nom de l'opération.

Le tag <body> possède plusieurs caractéristiques en Document/Literal wrapped :

- Le corps du message n'est composé que d'une seule partie
- Cette partie est encapsulée dans un élément dont le nom correspond au nom de l'opération invoquée
- Les éléments des paramètres ne possèdent aucun attribut

Le fichier WSDL

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://axis.test.jmdoudoux.com"
    xmlns:apachesoap="http://xml.apache.org/xml-soap"
    xmlns:impl="http://axis.test.jmdoudoux.com"
    xmlns:intf="http://axis.test.jmdoudoux.com"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<!--WSDL created by Apache Axis version: 1.3
Built on Oct 05, 2005 (05:23:37 EDT)-->
 <wsdl:types>
  <schema elementFormDefault="qualified"
      targetNamespace="http://axis.test.jmdoudoux.com"
      xmlns="http://www.w3.org/2001/XMLSchema">
   <element name="additionner">
    <complexType>
     <sequence>
      <element name="valeur1" type="xsd:int"/>
      <element name="valeur2" type="xsd:int"/>
     </sequence>
    </complexType>
   </element>
   <element name="additionnerResponse">
    <complexType>
     <sequence>
      <element name="additionnerReturn" type="xsd:long"/>
     </sequence>
    </complexType>
   </element>
  </schema>
 </wsdl:types>
   <wsdl:message name="additionnerResponse">
      <wsdl:part element="impl:additionnerResponse" name="parameters"/>
   </wsdl:message>
   <wsdl:message name="additionnerRequest">
      <wsdl:part element="impl:additionner" name="parameters"/>
   </wsdl:message>
   <wsdl:portType name="Calculer">
      <wsdl:operation name="additionner">
         <wsdl:input message="impl:additionnerRequest" name="additionnerRequest"/>
         <wsdl:output message="impl:additionnerResponse" name="additionnerResponse"/>
      </wsdl:operation>
   </wsdl:portType>
   <wsdl:binding name="CalculerSoapBinding" type="impl:Calculer">
      <wsdlsoap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
      <wsdl:operation name="additionner">
         <wsdlsoap:operation soapAction=""/>
         <wsdl:input name="additionnerRequest">
            <wsdlsoap:body use="literal"/>
         </wsdl:input>
         <wsdl:output name="additionnerResponse">
            <wsdlsoap:body use="literal"/>
         </wsdl:output>
      </wsdl:operation>
   </wsdl:binding>
   <wsdl:service name="CalculerService">
      <wsdl:port binding="impl:CalculerSoapBinding" name="Calculer">
         <wsdlsoap:address location="http://localhost:8080/TestWS/services/Calculer"/>
      </wsdl:port>
   </wsdl:service>
</wsdl:definitions>
```

Descripteur Axis

Exemple :

```
<service name="Calculer" provider="java:RPC" style="wrapped" use="literal">
  <operation name="additionner" qname="ns1:additionner"
      returnQName="ns1:additionnerReturn" returnType="xsd:long"
```

```
      soapAction="" xmlns:ns1="http://axis.test.jmdoudoux.com"
                    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <parameter qname="ns1:valeur1" type="xsd:int"/>
    <parameter qname="ns1:valeur2" type="xsd:int"/>
  </operation>
  <parameter name="allowedMethods" value="additionner"/>
  <parameter name="typeMappingVersion" value="1.2"/>
  <parameter name="wsdlPortType" value="Calculer"/>
  <parameter name="className" value="com.jmdoudoux.test.axis.Calculer"/>
  <parameter name="wsdlServicePort" value="Calculer"/>
  <parameter name="schemaQualified" value="http://axis.test.jmdoudoux.com"/>
  <parameter name="wsdlTargetNamespace" value="http://axis.test.jmdoudoux.com"/>
  <parameter name="wsdlServiceElement" value="CalculerService"/>
 </service>
```

La requête SOAP

Exemple :

```
<?xml version="1.0" encoding="UTF-8" ?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:q0="http://axis.test.jmdoudoux.com"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <q0:additionner>
      <q0:valeur1>20</q0:valeur1>
      <q0:valeur2>30</q0:valeur2>
    </q0:additionner>
  </soapenv:Body>
</soapenv:Envelope>
```

La réponse SOAP

Exemple :

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <additionnerResponse xmlns="http://axis.test.jmdoudoux.com">
      <additionnerReturn>50</additionnerReturn>
    </additionnerResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

Avantages

- Le contenu du corps du message est défini par un schéma permettant ainsi sa validation
- Le nom de l'opération est inclus dans la requête SOAP sous la forme d'un tag du nom de l'opération entre le tag <body>et les tags contenant les paramètres.
- Ce format est relativement proche du format RPC/Literal

Inconvénients

- Il n'est pas possible d'utiliser ce style avec des méthodes surchargées

54.3.6. Le choix du format à utiliser

Les différents formats (style et encodage) des services web ont tous des restrictions d'utilisation qui peuvent engendrer des limitations dans l'écriture des services ou forcer l'utilisation d'un format ou d'un autre.

54.3.6.1. L'utilisation de document/literal

Dans ce mode, le nom de l'opération n'est pas fourni dans le message : le mapping pour déterminer l'opération à invoquer repose donc sur les paramètres.

Il n'est donc pas possible d'invoquer le service ci-dessous avec le mode document/literal

Exemple :

```
public MonService {
  public void maMethode(int x, int y);
  public void maSecondeMethode(int x, int y);
}
```

54.3.6.2. L'utilisation de document/literal wrapped

Il pourrait être tentant de toujours utiliser le mode document/literal wrapped mais ce n'est pas toujours le meilleur choix :

- ce mode n'est pas supporté par tous les moteurs SOAP
- il n'est pas possible d'utiliser des opérations surchargées dans un service puisque le mapping de l'opération sur la méthode se fait sur le nom de la méthode. La classe ci-dessous ne peut pas être exposée sous la forme d'un service web invoqué via le mode document/literal wrapped.

Exemple :

```
public MonService {
  public void maMethode(int x, int y);
  public void maMethode(int x);
}
```

Remarque : WSDL 2.0 interdit l'utilisation des opérations surchargées.

54.3.6.3. L'utilisation de RPC/Literal

Comme le mode document/literal ne contient pas le nom de l'opération à invoquer, il y a des cas où il faut utiliser le mode document/literal wrapped ou un des deux modes RPC/encoded ou RPC/literal.

Exemple :

```
public MonService {
  public void maMethode(int x, int y);
  public void maMethode(int x);
  public void maSecondeMethode(int x, int y);
}
```

L'exemple ci dessus ne peut pas être invoqué ni en document/literal ni en document/literal wrapped.

Comme le mode RPC/encoded n'est pas WS-I basic profile compliant, il ne reste que le mode RPC/Literal

54.3.6.4. L'utilisation de RPC/encoded

Le mode RPC/encoded n'est pas WS-I Basic Profile compliant mais il est parfois nécessaire de l'utiliser. Ce mode est le seul qui puisse prendre en charge un graphe d'objets qui peut contenir plusieurs fois la même référence.

Exemple :

```
<complexType name="MonElement">
    <sequence>
        <element name="nom" type="xsd:string"/>
        <element name="partie1" type="MonElement" xsd:nillable="true"/>
        <element name="partie2" type="MonElement" xsd:nillable="true"/>
    </sequence>

</complexType>
```

RPC/Encoded utilise l'attribut id pour donner un identifiant à un élément et utilise un attribut href pour y faire référence.

Exemple :

```
<element1>
    <name>nom1</name>
    <partie1 href="1234"/>
    <partie2 href="1234"/>
</element1>
<element2 id="1234">
    <name>nom2</name>
    <partie1 xsi:nil="true"/>
    <partie2 xsi:nil="true"/>
</element2>
```

Dans le style literal, il n'y pas de moyen de faire une référence sur un objet déjà présent dans le graphe : la seule solution c'est de le dupliquer, ce qui va poser des soucis au consommateur du service.

54.4. Des conseils pour la mise en oeuvre

Avant de développer des services web, il faut valider la solution choisie avec un POC (Proof Of Concept) ou un prototype. Lors de ces tests, il est important de vérifier l'interopérabilité notamment si les services web sont consommés par différentes technologies.

Le choix du moteur SOAP est aussi très important notamment vis à vis du support des spécifications, des performances, de la documentation, ...

54.4.1. Les étapes de la mise en oeuvre

La mise en oeuvre de services web suit plusieurs étapes.

Etape 1 : définition des contrats des services métier

Cette étape est une phase d'analyse qui va définir les fonctionnalités proposées par chaque service pour répondre aux besoins

Etape 2 : identification des services web

Cette étape doit permettre de définir les contrats techniques des services web à partir des services métier définis dans l'étape précédente. Un service métier peut être composé d'un ou plusieurs services web.

La réalisation de cette étape doit tenir compte de plusieurs contraintes :

* Penser forte granularité / faible couplage
* Ternir compte de contraintes techniques
* Préférer les services web indépendants du contexte client

L'invocation d'un service est coûteuse notamment à cause du mapping objet/xml et xml/objet réalisé à chaque appel. Cette règle est vraie pour toutes les invocations de fonctionnalités distantes mais encore plus avec les services web. Il est donc préférable de limiter les invocations de méthodes d'un service web en proposant des fonctionnalités à forte granularité. Par exemple, il est préférable de définir une opération qui permet d'obtenir les données d'une entité plutôt que de proposer autant d'opérations que l'entité possède de champs. Ceci permet de réduire le nombre d'invocations de service web et réduit le couplage entre la partie front-end et back-end.

La définition des services web doit tenir compte de contraintes techniques liées aux performances ou à la consommation de ressources. Par exemple, si le temps de traitement d'un service web est long, il faudra prévoir son invocation de façon asynchrone ou si les données retournées sont des binaires de taille importante, il faudra envisager d'utiliser le mécanisme de pièces jointes (attachment).

Il est préférable de définir des services web qui soit stateless (ne reposant pas par exemple sur une utilisation de la session http). Ceci permet de déployer les services web dans un cluster ou la réplication de session sera inutile.

Etape 3 : écriture des services web

Cette étape est celle du codage proprement dit des services web.

Deux approches sont possibles :

* écriture du wsdl en premier (contract first) : des outils du moteur Soap sont utilisés pour gérer le code des services web à partir du wsdl. Face à la complexité de la rédaction du wsdl, cette approche n'est toujours privilégiée.
* écriture de la classe et génération du wsdl (code first) : chaque service est implémenté sous la forme d'une ou plusieurs classes et c'est le moteur Soap utilisé qui va générer le wsdl correspondant en se basant sur la description de la classe et des méta données.

Etape 4 : déploiement et tests

Les services web doivent être packagées et déployées généralement dans un serveur d'applications ou un conteneur web.

Pour tester les services web, il est possible d'utiliser des outils fournit par l'IDE utilisé ou d'utiliser des outils tiers comme SoapUI qui propose de très nombreuses fonctionnalités pour les tests des services web allant de la simple invocation à l'invocation de scénarios complexes et de tests de charges.

Etape 5 : consommation des services web par les applications clientes

Il faut utiliser les outils du moteur Soap utilisé par l'application cliente pour générer les classes nécessaires à l'invocation des services web et utiliser ces classes dans l'application. C'est généralement le moment de faire quelques adaptations pour permettre une bonne communication entre le client et le serveur.

54.4.2. Quelques recommandations

Afin de maximiser la portabilité d'un service web, il faut essayer de suivre quelques recommandations.

Il ne faut pas se lier à un langage de programmation :

* n'utiliser que des types communs : int, float, String, Date, ...
* ne pas utiliser de types spécifiques : Object, DataSet, ...
* éviter la composition d'objets
* utiliser un tableau ou des collections typées avec un generic plutôt qu'une collection non typée

Il faut éviter la surchage des méthodes.

Il faut éviter de transformer une classe en service web (notamment en utilisant des annotations) : il est recommandé de définir une interface qui va contenir le contrat du service de son implémentation. Cette remarque appliquée en POO doit aussi s'appliquer pour les services web.

54.4.3. Les problèmes liés à SOAP

SOAP est assez complexe et sa mise en oeuvre dépend de l'implémentation dans la technologie utilisée côté consommateur et fournisseur de services web. Il en résulte des problèmes d'interopérabilités alors qu'un des but de SOAP est pourtant de s'en affranchir.

Il existe plusieurs types de problèmes :

Les problèmes liés aux versions de SOAP

Les versions SOAP 1.1 et 1.2 étant incompatibles, cela peut entrainer des problèmes de compatibilité si les implémentations des moteurs SOAP utilisées ne supportent que des versions différentes de SOAP.

Ceci est notamment le cas si l'implémentation du moteur SOAP est assez ancienne.

Les problèmes liés aux modèles de messages

Un message Soap peut être encodé selon plusieurs modèles : le modèle le plus ancien (RPC) est abandonné au profit du modèle Document.

Cela peut introduire des problèmes d'incompatibilité notamment entre des services web existant et des consommateurs plus récents ou vice et versa.

54.5. Les API Java pour les services web

Java propose un ensemble d'API permettant la mise en oeuvre des services web.

API	Rôle
JAXP	API pour le traitement de documents XML : analyse en utilisant SAX ou DOM et transformation en utilisant XSLT.
JAX-RPC	API pour le développement de services web utilisant SOAP avec le style RPC
JAXM	API pour le développement de services utilisant des messages XML orientés documents
JAXR	API pour permettre un accès aux annuaires de référencement de services web
JAXB	API et outils pour automatiser le mapping d'un document XML avec des objets Java
StaX	API pour le traitement de documents XML
SAAJ	API pour permettre la mise en oeuvre des spécifications SOAP with Attachment
JAX-WS	API pour le développement grâce à des annotations de services web utilisant SOAP avec le style Document

L'API de base pour le traitement de document XML avec Java est JAXP. JAXP regroupe un ensemble d'API pour traiter des documents XML avec SAX et DOM et les modifier avec XSLT. Cette API est indépendante de tout parseur. JAXP est détaillé dans le chapitre «Java et XML».

D'autres API sont spécifiques au développement de service web :

- JAX-RPC (JSR-101) : permet l'appel de procédures distantes en utilisant SOAP (Remote Procedure Call)
- JAXM (JSR-67) : permet l'envoie de messages (en utilisant SAAJ)
- JAXR : permet l'accès au service de registre de façon standard (UDDI)

* SAAJ (SOAP with Attachment API for Java) : permet l'envoi et la réception de messages respectant les normes SOAP et SOAP with Attachment

54.5.1. JAX-RPC

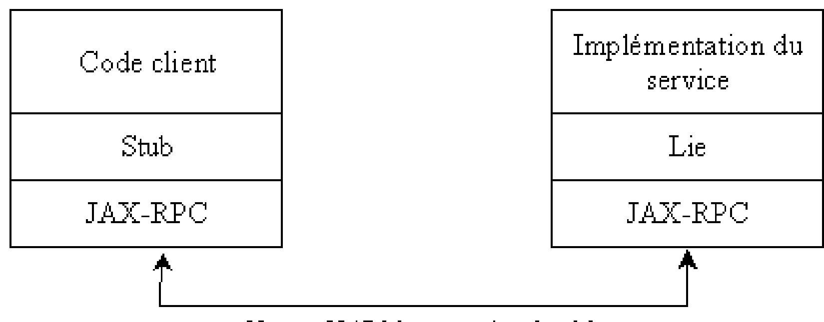
JAX-RPC est l'acronyme de Java API for XML-Based Remote Procedure Calls. Cette API permet la mise en oeuvre de services web utilisant SOAP aussi bien côté fournisseur que consommateur : elle permet l'appel de méthodes distantes et la réception de leur réponse en utilisant SOAP 1.1 et HTTP 1.1.

Cette API a été développée par le JCP sous la JSR 101.

Cette API propose de masquer un grand nombre de détails de l'utilisation de SOAP notamment en ce qui concerne le codage en XML du message et ainsi de rendre cette API facile à utiliser.

L'utilisation de JAX-RPC est similaire à celle de RMI : le code du client appel les méthodes à partir d'un objet local nommé stub. Cet objet se charge de dialoguer avec le serveur et de coder et décoder les messages SOAP échangés.

Côté serveur, un objet similaire nommé tie permet de réaliser le même type d'opération côté serveur.



La principale différence entre RMI et les services web est que RMI ne peut être utilisé qu'avec Java alors que les services web sont interopérables grâce à XML. Ainsi un client écrit en Java peut utiliser un service web développé avec.Net et vice et versa.

La spécification JAX-RPC définit précisément le mapping entre les types Schema et les types Java.

Type Schema	Type Java
xsd:boolean	boolean
xsd:short	short
xsd:int	int
xsd:long	long
xsd:integer	BigInteger
xsd:float	float
xsd:double	double
xsd:decimal	BigDecimal
xsd:date	java.util.calendar
xsd:time	java.util.calendar
xsd:datetime	java.util.calendar
xsd:base64Binary	byte[]
xsd:hexBinary	byte[]

Les types primitifs qui sont nillable sont mappés sur leurs wrappers Java correspondants.

Les types complexes sont mappés sur des Java beans.

L'API JAX-RPC est regroupée dans plusieurs sous packages du package javax.xml.rpc

54.5.1.1. La mise en oeuvre côté serveur

L'écriture d'un service web avec JAX-RPC requiert plusieurs entités :

- une interface facultative qui décrit le endpoint
- une implémentation du endpoint
- un ou plusieurs fichiers de description et configuration
- le wdsl du service web

L'utilisation de JAX-RPC côté serveur se fait en plusieurs étapes :

1. Définition de l'interface du service (écrite manuellement ou générée automatiquement par un outil à partir de la description du service (WSDL)).

Exemple :

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface MonWS extends Remote {
  public String getMessage(String nom) throws RemoteException;

}
```

Cette interface doit étendre l'interface java.rmi.Remote.

Toutes les méthodes définies dans l'interface doivent au minimum déclarer la possibilité de lever une exception de type java.rmi.RemoteException. Chaque méthode peut aussi déclarer d'autres exceptions dans sa définition du moment que ces exceptions héritent de la classe java.lang.Exception.

Les méthodes peuvent sans restriction utiliser des types primitifs et l'objet String pour les paramètres et la valeur de retour. Pour les autres types, il existe dans les spécifications une liste minimale prédéfinie de ceux utilisables.

Une implémentation particulière peut cependant proposer le support d'autres types. Par exemple, l'implémentation de référence propose le support de la plupart des classes de l'API Collection : ArrayList, HashMap, HashTable, LinkedList, TreeMap, TreeSet, Vector, ... Attention cependant dans ce cas, à la perte de la portabilité lors de l'utilisation d'une autre implémentation.

2. Ecriture de la classe d'implémentation du service

C'est une simple classe Java qui implémente l'interface définie précédemment.

Exemple :

```
public class MonWS_Impl implements MonWS {
  public String getMessage(String nom) {
    return new String("Bonjour " + nom);
  }

}
```

Cette implémentation doit obligatoirement implémenter l'interface définie précédemment et posséder un constructeur sans paramètre : dans l'exemple, celui ci sera généré lors de la compilation car il n'y a pas d'autre constructeur défini.

Il est inutile dans l'implémentation des méthodes de déclarer la levée de l'exception de type RemoteException. C'est lors de l'invocation de la méthode par JAX-RPC que cette exception pourra être levée.

3. Déploiement du service

Le déploiement dépend du moteur Soap utilisé et implique généralement la création d'un fichier de mapping entre l'url et la classe correspondante.

54.5.1.2. La mise en oeuvre côté client

JAX-RPC peut aussi être utilisée pour consommer un service web dans un client.

L'invocation de méthodes côté client se fait de manière synchrone avec JAX-RPC : le client fait appel au service et se met en attente jusqu'à la réception de la réponse

Cette invocation du service peut alors être faite selon trois modes :

- Un stub généré
- Un proxy dynamique
- Dynamic Invocation Interface

Un proxy dynamique met en oeuvre un mécanisme proche de celui utilisé par RMI : le client accède à un service distant en utilisant un stub. Le stub sert de proxy : il implémente l'interface du service et se charge des appels au service en utilisant le protocole SOAP lors de l'invocation de ses méthodes.

Ce proxy est généré par un compilateur dédié qui va utiliser le WSDL pour générer le proxy, notamment le portType pour définir l'interface de l'objet et le binding et port pour connaître les paramètres d'appel du service.

Le proxy généré est responsable de la transformation des invocations de méthodes en requête Soap et de la transformation du message Soap en réponse en objet selon les indications fournies dans le wsdl. En cas d'erreur, le message Soap de type fault est transformé en une exception.

La suite de cette section sera développée dans une version future de ce document

54.5.2. JAXM

JAXM est l'acronyme de Java API for XML Messaging. Cette API permet le développement de services utilisant des messages XML orientés documents.

JAXM a été développé sous la JSR-067.

Les classes de cette API sont regroupées dans le package javax.xml.messaging.

JAXM met en oeuvre SOAP 1.1 et SAAJ

54.5.3. JAXR

L'API JAXR (Java API for XML Registries) propose de standardiser les accès aux registres dans lesquels sont recensés les services web. JAXR permet notamment un accès aux registres de type UDDI ou ebXML.

Une implémentation de cette spécification doit être proposée par un fournisseur

Elle est incluse dans deux packages :

- javax.xml.registry : classes et interface de base (Connection, Query, LifeCycleManager, ...)
- javax.xml.registry.infomodel : interfaces qui décrivent les informations du modèle stockées dans un registre

Le support de l'accès aux registres de type ebXML est facultatif.

54.5.4. SAAJ

L'API SAAJ (SOAP with Attachment API for Java) permet l'envoi et la réception de messages respectant les normes SOAP 1.1 et SOAP with attachments : cette API propose un niveau d'abstraction assez élevé permettant de simplifier l'usage de SOAP.

Les classes de cette API sont regroupées dans le package javax.xml.soap.

Initialement, cette API était incluse dans JAXM. Depuis la version 1.1, elles ont été séparées.

SAAJ propose des classes qui encapsulent les différents éléments d'un message SOAP : SOAPMessage, SOAPPart, SOAPEnvelope, SOAPHeader et SOAPBody.

Tous les échanges de messages avec SOAP utilisent une connexion encapsulée dans la classe SOAPConnection. Cette classe permet la connexion directe entre l'émetteur et le receveur du ou des messages.

54.5.5. JAX-WS

JAX-WS (Java API for XML based Web Services) est une nouvelle API, mieux architecturée, qui remplace l'API JAX-RPC 1.1 mais n'est pas compatible avec elle. Il est fortement recommandé d'utiliser le modèle de programmation proposé par JAX-WS notamment pour les nouveaux développements.

Elle propose un modèle de programmation pour produire (côté serveur) ou consommer (côté client) des services web qui communiquent via des messages XML de type SOAP.

Elle a pour but de faciliter et simplifier le développement des services web notamment grâce à l'utilisation des annotations. JAX-WS fournit les spécifications pour le coeur du support des services web pour la plate-forme Java SE et Java EE.

JAX-WS a été spécifié par la JSR 224 : Java API for XML-Based Web Services (JAX-WS) 2.0.

JAX-WS permet la mise en oeuvre de plusieurs spécifications :

- JAX-WS respecte le standard WS-I Basic Profile version 1.1.
- JAX-WS propose un support pour SOAP 1.1 et 1.2
- JAX-WS permet le développement de services web orientés RPC (literal) ou orienté documents (literal/encoded/literal wrapped)

JAX-WS repose sur plusieurs autres JSR :

- JSR 181 (Web Services MetaData for the Java Platform) : propose un ensemble d'annotations qui permettent de définir les services web

* JSR 109 et JSR 921 (Implementing Enterprise Web Services) : décrit comment déployer, gérer et accéder aux services web via un serveur d'applications
* JSR 183 (Web Services Message Security APIs) : décrit la sécurisation des messages SOAP

Le fournisseur de l'implémentation de JAX-WS utilise les spécifications de la JSR 921 pour générer les fichiers de configurations et de déploiement à partir des annotations et d'éventuelles méta-données.

JAX-WS utilise JAXB 2.0 et SAAJ 1.3.JAXB propose une API et des outils pour automatiser le mapping un document XML et des objets Java. A partir d'une description du document XML (Schéma XML ou DTD), des classes sont générées pour effectuer automatiquement l'analyse du document ML et le mapping de ce dernier dans des objets Java.

JAX-WS peut être combiné avec d'autres spécifications comme les EJB 3 par exemple.

54.5.5.1. La mise en oeuvre de JAX-WS

JAX-WS est une spécification : pour la mettre en oeuvre, il faut utiliser une implémentation.

L'implémentation de référence de JAX-WS est le projet Metro développé par la communauté du projet GlassFish. Il existe d'autres implémentations notamment Axis 2 qui propose son propre modèle de programmation mais propose aussi un support de JAX-WS.

Le développement d'un service web en Java avec JAX-WS débute par la création d'une classe annotée avec @WebService du package javax.jws. La classe ainsi annotée définit le endpoint du service web.

Le service endpoint interface (SEI) est une interface qui décrit les méthodes du service : celles-ci correspondent aux opérations invocables par un client.

Il est possible de préciser explicitement le SEI en utilisant l'attribut endpointInterface de l'annotation @WebService

54.5.5.2. La production de service web avec JAX-WS

Par rapport à JAX-RPC, l'utilisation de JAX-WS est plus simple : un service web peut être basiquement définit en utilisant une classe de type POJO avec des annotations.

La classe d'implémentation du service est donc très simple : un simple POJO avec des annotations. Il n'y a pas besoin d'implémenter une interface particulière de l'API ni de déclarer une exception dans les méthodes.

Avec JAX-WS, la définition d'un service web et de ces opérations se fait en utilisant des annotations soit dans une interface qui décrit le service soit directement dans la classe d'implémentation.

Ni côté client ni côté serveur,le développeur n'a besoin de manipuler le contenu des messages Soap. Ceci est cependant possible pour des besoins très spécifiques.

Les annotations fournissent des méta-données exploitées par le moteur Soap pour générer le code des traitements sous jacents. Le développeur est ainsi déchargé de la plomberie et peut se concentrer sur les traitements métiers qui représentent la plus value du service.

Le développement d'un service web avec JAX-WS requiert plusieurs étapes :

* coder la classe qui encapsule le service
* compiler la classe
* utiliser lacommande wsgen pour générer les fichiers requis pour le déploiement (schémas, wsdl, classes, ...)
* packager le service dans un fichier.war
* déployer le war dans un conteneur

Pour définir un endpoint avec JAX-WS, il a plusieurs contraintes :

* la classe qui encapsule le endpoint doit être public, non static, non final, non abstract, et être annotée avec @WebService
* elle doit avoir un constructeur par défaut (sans paramètre)
* il est recommandé de définir explicitement l'interface du SEI
* les méthodes exposées par le service web doivent être public, non static, non final, et être annotées avec @WebMethod
* le type des paramètres et de la valeur de retour de ces méthodes doivent être supportés par JAXB

Exemple :

```
import javax.jws.WebService;
import javax.jws.WebMethod;
@WebService
public class MonService
{
  @WebMethod
  public
String saluer()
  {
    return "Bonjour";
  }
}
```

La classe qui encapsule le endpoint du service peut définir des méthodes annotées avec @PostConstruct et @PreDestroy pour définir des traitements liés au cycle de vie du service. Ces méthodes sont invoquées par le conteneur respectivement avant la première utilisation de la classe et avant le retrait du service.

Il faut compiler la classe et utiliser l'outil wsgen pour générer les classes et fichiers requis pour l'exécution du service web.

L'outil wsgen doit être utilisé pour générer les classes utiles pour permettre d'exposer le service web : celles-ci concernent essentiellement des classes qui utilisent JAXB pour mapper le contenu du message avec un objet et vice et versa. Il permet aussi de générer le wsdl et les schémas XML des messages.

La syntaxe est de la forme :

```
wsgen [options] <sei>
```

<sei> est le nom pleinement qualifié de classe d'implémentation du SEI.

Option	Rôle
-classpath <path> -cp <path>	Spécifier le classpath
-d <directory>	Préciser le répertoire qui va contenir les classes générées
-help	Afficher l'aide
-keep	Conserver les fichiers générés
-r <directory>	Préciser le répertoire qui va contenir les fichiers de ressources générés (wsdl, ...)
-s <directory>	Préciser le répertoire qui va contenir les fichiers sources générés
-verbose	Activer le mode verbeux
-version	Afficher la version
-wsdl[:protocol]	Demander la génération du wsdl : ce fichier n'est pas utilisé à l'exécution mais il peut être consulté par le développeur pour vérification. Le protocole est facultatif : il permet de préciser la version SOAP qui sera utilisé (les valeurs possibles sont soap1.1 et soap1.2)
-servicename <name>	Définir la valeur de l'attribut name du tag <wsdl:service> lorsque l'option -wsdl est utilisée
	Définir la valeur de l'attribut name du tag <wsdl:port> lorsque l'option -wsdl est utilisée

-portname <name>	

Une tache Ant est proposée pour invoquer wsgen via cet outil de build.

Il faut packager le service dans une webapp avec les fichiers compilés.

Il faut déployer le service dans un conteneur web ou un serveur d'applications. Au déploiement, JAX-WS va créer les différentes classes requises pour l'utilisation du service web (celles encapsulant les messages) si celles-ci ne sont pas présentes.

Pour voir le wsdl du service il faut utiliser l'url :

```
http://localhost:8080/helloservice/hello?wsdl
```

JAX-WS utilise JAXB-2.0 pour le mapping entre les objets et XML : les objets échangés via les services web peuvent utiliser les annotations de JAXB pour paramétrer finement certains éléments du message SOAP.

Exemple :

```
package com.jmdoudoux.test.ws;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebService;

@WebService()
public class PersonneWS {

  @WebMethod(operationName = "Saluer")
  public String Saluer(@WebParam(name = "personne") final Personne personne) {
    return "Bonjour " + personne.getNom() + " "+personne.getPrenom();
  }

}
```

Exemple :

```
package com.jmdoudoux.test.ws;

import java.util.Date;

public class Personne {

  private String nom;
  private String prenom;
  private Date dateNaiss;

  public Personne() {
    super();
  }

  public Personne(String nom, String prenom, Date dateNaiss) {
    super();
    this.nom = nom;
    this.prenom = prenom;
    this.dateNaiss = dateNaiss;
  }

  public synchronized String getNom() {
    return nom;
  }

  public synchronized void setNom(String nom) {
    this.nom = nom;
  }

  public synchronized String getPrenom() {
```

```
    return prenom;
  }

  public synchronized void setPrenom(String prenom) {
    this.prenom = prenom;
  }

  public synchronized Date getDateNaiss() {
    return dateNaiss;
  }

  public synchronized void setDateNaiss(Date dateNaiss) {
    this.dateNaiss = dateNaiss;
  }
}
```

Le WSDL généré définit l'élément avec un nom dont la première lettre est en minuscule.

Exemple :

```
<?xml version='1.0' encoding='UTF-8'?>
<!-- Published by JAX-WS RI at http://jax-ws.dev.java.net. RI's version is
JAX-WS RI 2.1.7-hudson-48-. -->
<xs:schema xmlns:tns="http://ws.test.jmdoudoux.com/"
              xmlns:xs="http://www.w3.org/2001/XMLSchema"
              version="1.0" targetNamespace="http://ws.test.jmdoudoux.com/">

<xs:element name="Saluer" type="tns:Saluer" />

<xs:element name="SaluerResponse" type="tns:SaluerResponse" />

<xs:complexType name="Saluer">
<xs:sequence>
<xs:element name="personne" type="tns:personne" minOccurs="0" />
</xs:sequence>
</xs:complexType>

<xs:complexType name="personne">
<xs:sequence>
<xs:element name="dateNaiss" type="xs:dateTime" minOccurs="0" />
<xs:element name="nom" type="xs:string" minOccurs="0" />
<xs:element name="prenom" type="xs:string" minOccurs="0" />
</xs:sequence>
</xs:complexType>

<xs:complexType name="SaluerResponse">
<xs:sequence>
<xs:element name="return" type="xs:string" minOccurs="0" />
</xs:sequence>
</xs:complexType>
</xs:schema>
```

En utilisant l'annotation @XmlType, il est possible de forcer le nom de l'élément généré dans le schéma

Exemple :

```
package com.jmdoudoux.test.ws;

import java.util.Date;

import javax.xml.bind.annotation.XmlType;

@XmlType(name = "Personne")
public class Personne {

  private String nom;
  private String prenom;
  private Date dateNaiss;
```

```
    ...
}
```

Le WSDL généré définit l'élément avec un nom dont la première lettre est en majuscule.

Exemple :

```
<?xml version='1.0' encoding='UTF-8'?>
<!-- Published by JAX-WS RI at http://jax-ws.dev.java.net. RI's version is
JAX-WS RI 2.1.7-hudson-48-. -->
<xs:schema xmlns:tns="http://ws.test.jmdoudoux.com/"
              xmlns:xs="http://www.w3.org/2001/XMLSchema"
              version="1.0" targetNamespace="http://ws.test.jmdoudoux.com/">

<xs:element name="Saluer" type="tns:Saluer" />

<xs:element name="SaluerResponse" type="tns:SaluerResponse" />

<xs:complexType name="Saluer">
<xs:sequence>
<xs:element name="personne" type="tns:Personne" minOccurs="0" />
</xs:sequence>
</xs:complexType>

<xs:complexType name="Personne">
<xs:sequence>
<xs:element name="dateNaiss" type="xs:dateTime" minOccurs="0" />
<xs:element name="nom" type="xs:string" minOccurs="0" />
<xs:element name="prenom" type="xs:string" minOccurs="0" />
</xs:sequence>
</xs:complexType>

<xs:complexType name="SaluerResponse">
<xs:sequence>
<xs:element name="return" type="xs:string" minOccurs="0" />
</xs:sequence>
</xs:complexType>
</xs:schema>
```

54.5.5.3. La consommation de services web avec JAX-WS

Côté client, un proxy est généré pour faciliter l'invocation des opérations du service web.

Dans la partie cliente, un objet de type proxy est généré pour faciliter l'invocation et la consommation des services web. Les classes de ce proxy sont générées par l'outil wsimport à la demande du développeur à partir du wsdl.

Pour développer un client qui consomme le service web, il y a plusieurs étapes :

- utiliser l'outil wsimport pour générer les classes du proxy
- écrire le code des traitements en utilisant le proxy
- compiler toutes les classes
- exécuter le client

La suite de cette section sera développée dans une version future de ce document

54.5.5.4. Les handlers

Les handlers proposent un mécanisme pour fournir des traitements particuliers exécutés par le moteur Soap pour permettre d'agir sur les messages de type requête ou réponse. JAX-WS propose deux types de handlers selon la source des données à obtenir ou modifier dans le message :

- Protocol handler : ce type de handler est dédié à un protocole particulier par exemple Soap. Il permet d'obtenir des informations ou d'en modifier dans toutes les parties du message
- Logical Handler : ce type de handler est indépendant du protocole utilisé par le message en permettant une modification du corps du message à partir de son contexte JAXB.

Les handlers sont généralement utiliser pour traiter des informations particulières du message Soap

Les handlers pour le protocole SOAP doivent hériter de la classe javax.xml.ws.handler.soap.SOAPHandler. La classe SOAPMessageContext propose des méthodes pour permettre un accès au contenu du message encapsulé dans un objet de type SOAPMessage. Le contenu du message peut alors être manipulé avec l'API SAAJ.

Les logical handler doivent hériter de la classe javax.xml.ws.handler.LogicalHandler. Ils permettent un accès au contenu du message qui correspond pour un message de type SOAP au body. La classe LogicalMessageContext propose des méthodes pour permettre un accès au contenu du message encapsulé dans un objet de type LogicalMessage.

La suite de cette section sera développée dans une version future de ce document

54.5.6. La JSR 181 (Web Services Metadata for the Java Platform)

La JSR 181 propose une spécification pour permettre le développement de services web en utilisant des POJO et des annotations.

La JSR 181 a pour but de définir un modèle de programmation pour faciliter le développement des services web. Ce modèle repose essentiellement sur les annotations : ceci permet de définir les services web sans avoir à connaître les détails de l'implémentation qui sera mise en oeuvre.

Les annotations proposées permettent un contrôle assez fin sur la façon dont un service web va être exposé et invoqué.

La JSR 181 est une spécification dont le but est de fournir un standard pour la déclaration de services web en proposant :

- Un modèle standard pour le développement de services web en utilisant des annotations
- De masquer les détails de l'implémentation
- D'assurer la maintenabilité et l'interopérabilité

Chaque implémentation de cette JSR doit fournir des fonctionnalités pour permettre d'exécuter les classes annotées dans un environnement d'exécution pour les services web.

La mise en oeuvre suit plusieurs étapes :

- Ecriture de la classe qui contient les fonctionnalités à exposer
- Annoter la classe ou son interface
- Exploitation par une implémentation des annotations de la classe pour générer des schémas XML, le WSDL et d'autres fichiers requis pour le déploiement

Exemple :

```
import javax.jws.WebService;
import javax.jws.WebMethod;
@WebService
public class MonService
{
  @WebMethod
  public
String saluer()
  {
    return "Bonjour";
  }
}
```

Il existe plusieurs contraintes dont il faut tenir compte lors de l'implémentation du service. La classe de l'implémentation doit :

- être public,
- non final,
- non abstract,
- avoir un constructeur par défaut

Par défaut, toutes les méthodes public sont exposées sous la forme d'une opération et ne doivent utiliser que des paramètres respectant ceux définis dans JAX-RPC 1.1. Les méthodes héritées sont aussi exposées sauf celles héritées de la classe Object.

54.5.6.1. Les annotations définies

Les annotations sont utilisées dans la classe d'implémentation ou dans l'interface d'un service web.

Toutes les annotations de la JSR 181 sont définies dans le package javax.jws.

Ces annotations sont exploitées au runtime.

Attention : plusieurs implémentations fournissent en plus des annotations de la JSR 181 des annotations qui leur sont propres. Même si elles sont pratiques, elle limite la portabilité des services web à s'exécuter dans un autre moteur Soap (exemple : @EnableMTOM, @ServiceProperty, @ServicesProperties dans XFire).

54.5.6.2. javax.jws.WebService

L'annotation javax.ws.WebService permet de définir une classe ou une interface comme étant l'interface du endpoint d'un service web.

L'annotation WebService est la seule annotation obligatoire pour développer un service web.

Si l'annotation est utilisée sur l'interface du service web (SEI), il faut aussi l'utiliser sur la classe d'implémentation en précisant l'interface via l'attribut endpointInterface.

Cette annotation s'utilise sur une classe ou une interface uniquement.

Attributs	Rôle
String name	le nom du service web utilisé dans l'attribut name de l'élément wsdl:portType du WSDL Par défaut, c'est le nom non qualifié de la classe
String targetNamespace	espace de nommage utilisé dans le wsdl

	Par défaut c'est le nom du package
String serviceName	le nom du service utilisé dans l'attribut name de l'élément wsdl:service du WSDL Par défaut, c'est le nom de la classe suffixée par "Service"
String wsdlLocation	url relative ou absolue du wsdl prédéfini
String endpointInterface	nom pleinement qualifié de l'interface du endpoint (SEI), ce qui permet de séparer l'interface de l'implémentation
String portName	Nom du port du service web utilisé dans l'attribut name de l'élément wsdl:port du WSDL

Exemple :

```
@WebService(
name = "BonjourWS",
targetNamespace = "http://www.jmdoudoux.fr/ws/Bonjour"
)
public class BonjourServiceImpl {
@WebMethod
public String saluer() {
return "Bonjour";
}
}
```

54.5.6.3. javax.jws.WebMethod

L'annotation javax.ws.WebMethod permet de définir une méthode comme étant une opération d'un service web.

Cette annotation s'utilise sur une méthode uniquement. La méthode sur laquelle cette annotation est appliquée doit être public.

Elle possède plusieurs attributs.

Attributs	Rôle
String operationName	nom utilisé dans l'élément wsdl:operation du message Par défaut: le nom de la méthode
String action	action associée à l'opération : utilisé comme valeur du paramètre SOAPAction
boolean exclude	booléen qui précise si la méthode doit être exposée ou non dans le service web. Cette propriété n'est utilisable que dans une classe et doit être le seul attribut de l'annotation. Par défaut : false

L'annotation WebMethod ne peut être utilisée que dans une classe ou une interface annotée avec @WebService.

Les paramètres de la méthode, sa valeur de retour et les exceptions qu'elle peut lever doivent obligatoirement respecter les spécifications relatives à ces entités dans les spécifications JAX-RPC 1.1.

54.5.6.4. javax.jws.OneWay

L'annotation javax.ws.OneWay permet de définir une méthode comme étant une opération d'un service web qui ne fournit pas de réponse lors de son invocation. Elle permet une optimisation à l'exécution qui évite d'attendre une réponse

qui ne sera pas fournie.

Cette annotation s'utilise sur une méthode uniquement : celle-ci ne doit pas avoir de valeur de retour ou lever une exception puisque dans ce cas, il y a une réponse de type SoapFault.

Cette annotation ne possède aucun attribut.

Exemple :

```
@WebService
public class MonService {

  @WebMethod
  @Oneway
  public void MonOperation() {
  }
};
```

54.5.6.5. javax.jws.WebParam

L'annotation javax.ws.WebParam permet de configurer comment un paramètre d'une opération sera mappé dans le message SOAP.

Cette annotation s'utilise uniquement sur un paramètre d'une méthode de l'implémentation du service.

Attributs	Rôle
String name	nom du paramètre utilisé dans le wsdl Par défaut: le nom du paramètre
Mode mode	mode d'utilisation du paramètre. Le type Mode est une énumération qui contient IN, OUT et INOUT Par défaut : IN
String targetNamespace	précise l'espace de nommage du paramètre dans les messages utilisant le mode document Par défaut : l'espace de nommage du service web
boolean header	booléen qui indique si la valeur du paramètre est contenue dans l'en-tête de la requête http plutôt que dans le corps Par défaut : false
String partName	Définit l'attribut name de l'élément <wsdl:part> des messages de type RPC et DOCUMENT/BARE

Cette annotation est pratique pour permettre d'utiliser le même paramètre dans plusieurs opérations d'un service web encodé en document literal.

54.5.6.6. javax.jws.WebResult

L'annotation javax.ws.WebResult permet de configurer comment une valeur de retour d'une opération sera mappée dans l'élément wsdl:part message SOAP.

Cette annotation s'utilise sur une méthode uniquement.

Attributs	Rôle
String name	nom de la valeur de retour utilisé dans le wsdl. Avec le style RPC, c'est l'attribut name de l'élément wsdl:part. Avec le style DOCUMENT, c'est le nom de l'élément dans la réponse Par défaut: return pour RPC et DOCUMENT/WRAPPED et le nom de la méthode suffixé par "Response" pour DOCUMENT/BARE
String targetNamespace	espace de nommage de la valeur de retour dans les messages utilisant le mode document Par défaut : l'espace de nommage du service web
boolean header	booléen qui indique si la valeur de retour est stockée dans l'en-tête de la requête http plutôt que du corps Par défaut : false
String partName	nom de la part des messages de type RPC et DOCUMENT/BARE (attribut name de l'élément wsdl:part) Par défaut : la valeur de l'attribut name

Cette annotation est pratique pour permettre d'utiliser la même valeur de retour dans plusieurs opérations d'un service web encodé en document literal.

54.5.6.7. javax.jws.soap.SOAPBinding

L'annotation javax.jws.soap.SOAPBinding permet de configurer comment le message et la réponse SOAP vont être encodés.

Cette annotation s'utilise sur une classe, une interface ou une méthode.

Attributs	Rôle
Style style	Définir le style d'encodage du message. Style est une énumération qui contient DOCUMENT et RPC. Par défaut: DOCUMENT
Use use	Définir le format du message. Use est une énumération qui contient ENCODED et LITERAL. Par défaut: LITERAL
ParameterStyle parameterStyle	Définir si les paramètres forment le contenu du message ou s'ils sont encapsulés par un tag du nom de l'opération à invoquer. ParameterStyle est une énumération qui contient BARE et WRAPPED. BARE ne peut être utilisé qu'avec le style DOCUMENT Par défaut: WRAPPED

Exemple :

```
@WebService
@SOAPBinding(style=Style.DOCUMENT, use=Use.LITERAL, parameterStyle=ParameterStyle.BARE)
public class MonService {

  @WebMethod

  public void MonOperation() {
  }
};
```

54.5.6.8. javax.jws.HandlerChain

La suite de cette section sera développée dans une version future de ce document

54.5.6.9. javax.jws.soap.SOAPMessageHandlers

La suite de cette section sera développée dans une version future de ce document

54.5.7. La JSR 224 (JAX-WS 2.0 Annotations)

La JSR 224 définit les annotations spécifiques à JAX-WS 2.0.

Toutes ces annotations sont dans le package javax.xml.ws.

54.5.7.1. javax.xml.ws.BindingType

L'annotation @BindingType permet de préciser le binding qui sera utilisé pour invoquer le service. Elle s'utilise sur une classe

Attributs	Rôle
value	Identifiant du binding à utiliser. Les valeurs possibles sont : SOAPBinding.SOAP11HTTP_BINDING, SOAPBinding.SOAP12HTTP_BINDING ou HTTPBinding.http_BINDING La valeur par défaut est SOAP11_HTTP_BINDING

54.5.7.2. javax.xml.ws.RequestWrapper

L'annotation @RequestWrapper permet de préciser la classe JAXB de binding qui sera utilisée dans la requête à l'invocation du service. Elle s'utilise sur une méthode

Attributs	Rôle

String className	Préciser le nom pleinement qualifié de la classe qui encapsule la requête (Obligatoire)
String localName	Définir le nom de l'élément dans le schéma qui encapsule la requête. Par défaut, c'est la valeur de l'attribut operationName de l'annotation WebMethod
String targetNamespace	l'espace de nommage. Par défaut, c'est le targetNamespace du SEI

54.5.7.3. javax.xml.ws.ResponseWrapper

L'annotation @ResponseWrapper permet de préciser la classe JAXB de binding qui sera utilisée dans la réponse à l'invocation du service. Elle s'utilise sur une méthode

Attributs	Rôle
String localName	Définir le nom de l'élément dans le schéma qui encapsule la réponse. Par défaut c'est le nom de l'opération défini par l'annotation @WebMethod concaténé à Response
String targetNamespace	Définir l'espace de nommage. Par défaut, c'est le targetNamespace du SEI
String ClassName	Préciser le nom pleinement qualifié de la classe qui encapsule la réponse (Obligatoire)

54.5.7.4. javax.xml.ws.ServiceMode

Cette annotation permet de préciser si le provider va avoir accès uniquement au payload du message (PAYLOAD) ou à l'intégralité du message (MESSAGE).

Elle s'utilise sur une classe qui doit obligatoirement implémenter un Provider.

Attributs	Rôle
Service.Mode value	Indiquer si le provider va avoir accès uniquement au payload du message (PAYLOAD) ou à l'intégralité du message (MESSAGE). La valeur par défaut est PAYLOAD

Exemple :

```
@ServiceMode(value=Service.Mode.PAYLOAD)
public class MonOperationProvider implements Provider<Source> {
  public Source invoke(Source source)
    throws WebServiceException {
    Source source = null;
    try {

      // code du traitement de la requete et generation de la reponse

    } catch(Exception e) {
      throw new WebServiceException("Erreur durant les traitements du Provider", e);
    }
    return source;
  }

}
```

54.5.7.5. javax.xml.ws.WebFault

Cette annotation s'utilise sur une classe qui encapsule une exception afin de personnaliser certains éléments de la partie Fault du message Soap. Elle s'utilise sur une exception levée par une opération.

Attributs	Rôle
String name	Le nom de l'élément fault Cet attribut est obligatoire
String targetNameSpace	Définir l'espace de nommage pour l'élément fault.
String faultBean	Nom pleinement qualifié de la classe qui encapsule l'exception

54.5.7.6. javax.xml.ws.WebEndpoint

Cette annotation permet de préciser le port name d'une méthode du SEI.

Elle s'utilise sur une méthode.

Attributs	Rôle
String name	Définir le nom qui va identifier de façon unique l'élément <wsdl:port> du tag <wsdl:service>

54.5.7.7. javax.xml.ws.WebServiceclient

La suite de cette section sera développée dans une version future de ce document

54.5.7.8. javax.xml.ws.WebServiceProvider

Cette annotation est à utiliser sur une implémentation d'un Provider

Elle s'utilise sur une classe qui héritent de la classe Provider.

Attributs	Rôle
String portName	nom du port du service (élément <wsdl:portName>)
String serviceName	nom du service (élément <wsdl:service>)
String targetNamespace	espace de nommage
String wsdlLocation	chemin du wsdl du service

Exemple :

```
@WebServiceProvider
public class MonOperationProvider implements Provider<Source> {
  public Source invoke(Source source)
    throws WebServiceException {
    Source source = null;
    try {

      // code du traitement de la requete et generation de la reponse

    } catch(Exception e) {
      throw new WebServiceException("Erreur durant les traitements du Provider", e);
    }
    return source;
  }

}
```

54.5.7.9. javax.xml.ws.WebServiceRef

L'annotation WebServiceRef permet de définir une référence sur un service web et éventuellement de permettre son injection.

Cette annotation est à utiliser dans un contexte Java EE.

Elle s'utilise sur une classe, une méthode (getter ou setter) ou un champ.

Attributs	Rôle
String name	nom JNDI de la ressource. Par défaut sur un champ c'est le nom du champ. Par défaut sur un getter ou un setter, c'est le nom de la propriété
Class type	type de la ressource. Par défaut sur un champ, c'est le type du champ. Par défaut sur un getter ou un setter, c'est le type de la propriété
String mappedName	nom spécifique au conteneur sur lequel le service est mappé (non portable)
Class value	classe du service qui doit étendre javax.xml.ws.Service
String wsdlLocation	chemin du wsdl du service

54.6. Les implémentations des services web

Il existe de nombreuses implémentations possibles de moteurs SOAP permettant la mise en oeuvre de services web avec Java notamment plusieurs solutions open source:

- Intégrées à la plate-forme Java EE 5.0 et Java SE 6.0
- JWSDP de Sun
- Axis et Axis 2 du projet Apache
- XFire
- CXF du projet Apache
- JBoss WS
- Metro du projet GlassFish
- ...

Malgré un effort de spécification tardif via JAX-WS, plusieurs implémentations utilisent une approche spécifique pour la mise en oeuvre et le déploiement de services web, ce qui rend le choix d'une de ces solutions délicat. Heureusement, toutes tendent à proposer un support de JAX-WS.

Même si les concepts sous jacents sont équivalents, quelle que soit l'implémentation utilisée, leur mise en oeuvre est très différente d'une implémentation à l'autre.

De plus, la plupart des solutions historiques sont relativement complexes à mettre en oeuvre car certains points techniques ne sont pas assez masqués par les outils (code à écrire, fichiers de configuration, descripteurs de déploiement, ...). Avec ces solutions, le développeur doit consacrer une part non négligeable de son temps à du code technique pour développer le service web.

JAX-WS propose une solution pour simplifier grandement le développement des services grâce à l'utilisation d'annotations qui évite d'avoir à écrire du code ou des fichiers pour la plomberie. JAX-WS, en tant que spécification, est implémentée dans plusieurs solutions.

54.6.1. Axis 1.0

Axis (Apache eXtensible Interaction System) est un projet open-source du groupe Apache diffusé sous la licence Apache 2.0 qui propose une implémentation d'un moteur de service web qui implémente le protocole SOAP : il permet de créer, déployer et de consommer des services web.

Son but est de proposer un ensemble d'outils pour faciliter le développement, le déploiement et l'utilisation des services web écrits en java. Axis propose de simplifier au maximum les tâches pour la création et l'utilisation des services web. Il permet notamment de générer automatiquement le fichier WSDL à partir d'une classe java et le code nécessaire à l'appel du service web.

Pour son utilisation, Axis 1.0 nécessite un J.D.K. 1.3 minimum et un conteneur de servlets (les exemples de cette section utilise Tomcat).

Le site officiel est à l'url http://ws.apache.org/axis

C'est un projet open source d'implémentation du protocole SOAP. Il est historiquement issu du projet Apache SOAP.

C'est un outil populaire qui de fait est la référence des moteurs de services web Open Source implémentant JAX-RPC en Java : son utilisation est répandue notamment dans des produits open source ou commerciaux.

La version 1.2 diffusé en mai 2005 apporte le support de l'encodage de type document/literal pour être compatible avec les spécifications WS-I Basic Profile 1.0 et JAX-RPC 1.1.

La version 1.3 est diffusée en octobre 2005

La version la plus récente est la 1.4, diffusée en avril 2006.

Attention : Axis 1.x n'est plus supporté au profit de Axis 2 qui possède lui aussi des versions 1.x.

Axis implémente plusieurs spécifications :

- JSR 101 : Java API for XML-Based RPC (JAX-RPC) 1.1
- JSR 67 : SOAP with Attachments API for Java Specification (SAAJ) 1.2
- Java API for XML Registries Specification (JAXR) 1.0

Axis permet donc la mise en oeuvre de :

- SOAP 1.1 et 1.2
- WSDL 1.1
- XML-RPC
- WS-I Basic Profile 1.1

Attention : Axis 1.0 n'est pas compatible avec

- JSR 109 Web Services for EE (WS4EE) 1.0
- JSR 224 Java API for XML-Based Web Services (JAX-WS) 2.0
- JSR 181 Web Service Metadata for the Java Platform
- JSR 222 Java Architecture for XML Binding (JAXB) 2.0

Axis génère le document wsdl du service web : pour accéder à ce document il suffit d'ajouter ?wsdl à l'url d'appel du service web.

L'interopérabilité entre Axis et .Net 1.x est assurée tant que les types utilisés se limitent aux primitives, aux chaînes de caractères, aux tableaux des types précédents et aux Java Beans composés uniquement des types précédents ou d'autres Java Beans.

L'interopérabilité entre Axis 1.4 et .Net 2.0 est bien meilleure. Par exemple, la gestion des objets Nullable dans .Net 2.0 est prise en compte (notamment pour les dates et types primitifs) : il n'est donc plus nécessaire d'utiliser une gestion particulière pour ces objets.

Les extensions sont mises en oeuvre au travers du mécanisme de handlers.

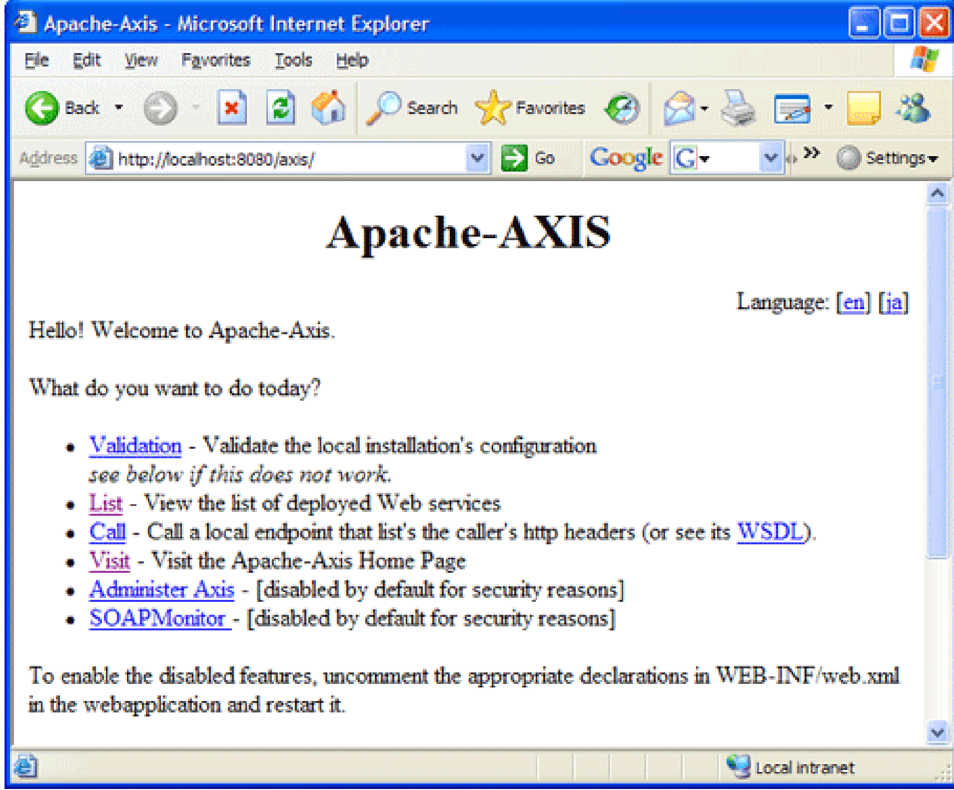
54.6.1.1. Installation

Il faut télécharger Axis 1.x (par exemple le fichier axis-bin-1_4.zip pour la version 1.4) sur le site et décompresser le contenu de l'archive dans un répertoire du système.

Axis s'utilise en tant qu'application web dans un conteneur web. Pour un environnement de développement avec Tomcat, le plus simple est de copier le répertoire axis contenu dans le sous répertoire webapps issu de la décompression dans le répertoire des applications web du conteneur (le répertoire webapps pour le serveur Tomcat) et de redémarrer le serveur.

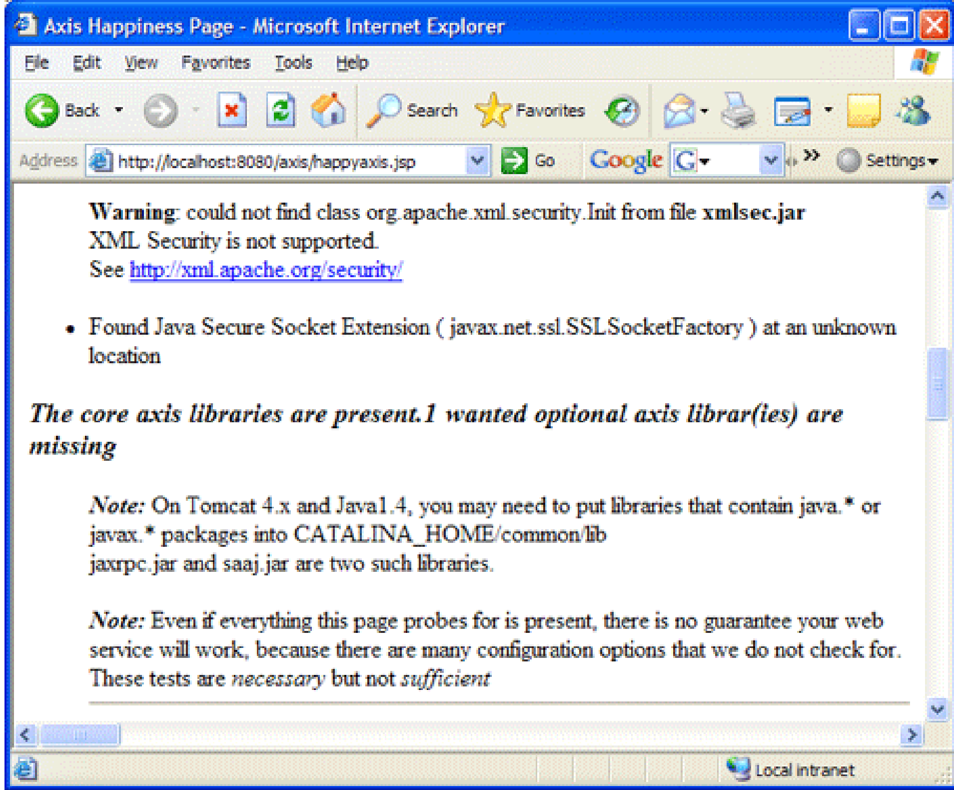
Pour vérifier la bonne installation, il suffit d'ouvrir un navigateur sur l'url de l'application web axis :

```
http://localhost:8080/axis/index.html
```



Un clic sur le lien « List » permet de voir quels sont les services web qui sont installés.

Un clic sur le lien « Validation » permet d'exécuter une JSP qui fait un état des lieux de la configuration du conteneur et des API nécessaires et optionnelles accessibles.



54.6.1.2. La mise en oeuvre côté serveur

Axis 1.x propose deux méthodes pour développer et déployer un service web :

- le déploiement automatique d'une classe java dont l'extension est .jws
- l'utilisation d'un fichier WSDD avec la classe d'implémentation

54.6.1.2.1. Mise en oeuvre côté serveur avec JWS

Axis propose une solution pour facilement et automatiquement déployer une classe java en tant que service web. Il suffit simplement d'écrire la classe, de remplacer l'extension .java en .jws (java web service) et de copier le fichier dans le répertoire de la webapp axis.

Remarque : il ne faut pas compiler le fichier .jws

54.6.1.2.2. Mise en oeuvre côté serveur avec un descripteur de déploiement

Cette solution est un peu moins facile à mettre en oeuvre mais elle permet d'avoir un meilleur contrôle sur le déploiement du service web.

Il faut écrire la classe java qui va contenir les traitements proposés par le service web.

Exemple :

```
public class MonServiceWebAxis2{
  public String message(String msg){
    return "Bonjour "+msg;
  }

}
```

Il faut compiler cette classe et mettre le fichier .class dans le répertoire WEB-INF/classes de la webapps axis.

Il faut créer le fichier WSDD qui va contenir la description du service web.

deployMonServiceWebAxis2.wsdd

Exemple :

```
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
            xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
  <service name="monServiceWebAxis2" provider="java:RPC">
           <parameter name="className" value="MonServiceWebAxis2"/>
    <parameter name="allowedMethods" value="*"/>
  </service>

</deployment>
```

Il faut ensuite déployer le service web en utilisant l'application AdminClient fournie par Axis.

Résultat :

```
C:\java\jwsdp-1.1\webapps\axis\WEB-INF\classes>java org.apache.axis.client.Admin
Client deployMonServiceWebAxis2.wsdd
- Processing file deployMonServiceWebAxis2.wsdd

-
<Admin>Done processing</Admin>
```

L'extension wsdd signifie WebService Deployment Descriptor.

C'est un document xml dont le tag racine est deployment.

Les informations relatives au service web sont définies dans le tag service qui possède plusieurs attributs notamment :

- name :
- provider :

Plusieurs informations doivent être fournies avec un tag parameter qui possède les attributs name et value :

- className : le nom pleinement qualifié de la classe d'implémentation du service web
- allowedMethods : précise les méthodes qui sont exposées. Le caractère étoile permet d'indiquer toutes les méthodes

54.6.1.3. Mise en oeuvre côté client

Pour faciliter l'utilisation d'un service web, Axis propose l'outil WSDL2Java qui génère automatiquement à partir d'un document WSDL des classes qui encapsulent l'appel à un service web. Grace à ces classes, l'appel d'un service web par un client ne nécessite que quelques lignes de code.

Résultat :

```
C:\java\jwsdp-1.1\webapps\axis\WEB-INF\classes>java org.apache.axis.wsdl.WSDL2Ja

va
http://localhost:8080/axis/services/monServiceWebAxis2?wsdl
```

L'utilisation de l'outil WSDL2Java nécessite une url vers le document WSDL qui décrit le service web. Il génère à partir de ce fichier plusieurs classes dans le package localhost. Ces classes sont utilisées dans le client pour appeler le service web.

Résultat :

```
C:\java\jwsdp-1.1\webapps\axis\WEB-INF\classes>java org.apache.axis.wsdl.WSDL2Ja
va http://localhost:8080/axis/services/monServiceWebAxis2?wsdl
```

Il faut utiliser les classes générées pour appeler le service web.

Exemple :

```
import localhost.MonServiceWebAxis2;
import localhost.*;

public class MonServiceWebAxis2Client{

  public static void main(String[] args) throws Exception{
    MonServiceWebAxis2Service locator = new MonServiceWebAxis2ServiceLocator();
    MonServiceWebAxis2 monsw = locator.getmonServiceWebAxis2();
    String s = monsw.message("Jean Michel");
    System.out.println(s);
  }

}
```

Résultat :

```
C:\java\jwsdp-1.1\webapps\axis\WEB-INF\classes>javac MonServiceWebAxis2client.java
C:\java\jwsdp-1.1\webapps\axis\WEB-INF\classes>java MonServiceWebAxis2Client

Bonjour Jean Michel
```

Axis propose une API regroupée dans le package org.apache.axis.client pour faciliter l'appel de service web par un client.

Exemple :

```
package com.jmdoudoux.test.axis;

import java.rmi.RemoteException;

import javax.xml.namespace.QName;
import javax.xml.rpc.ServiceException;

import org.apache.axis.client.Call;
import org.apache.axis.client.Service;

public class TestCalculerManuel {

  public static void main(String[] args) {
    Service service = new Service();
    Call call;

    try {
      call = (Call) service.createCall();
      String endpoint = "http://localhost:8080/TestWS/services/Calculer";

      call.setTargetEndpointAddress(endpoint);
      call.setOperationName(new QName("additionner"));
      long resultat = (Long) call.invoke(new Object[] { 10, 20 });

      System.out.println("resultat = " + resultat);

    } catch (ServiceException e) {
      e.printStackTrace();
    } catch (RemoteException e) {
      e.printStackTrace();
    }
  }

}
```

Résultat :

```
26 déc. 2006 11:22:32 org.apache.axis.utils.JavaUtils isAttachmentSupported
ATTENTION: Unable to find required classes (javax.activation.DataHandler
and javax.mail.internet.MimeMultipart). Attachment support is disabled.
resultat = 30
```

La classe Call permet l'invocation d'une méthode d'un service web. Une instance de cette classe est obtenue en utilisant la méthode createCall() d'un objet de type Service.

La méthode setTargetEndpointAddress() permet de préciser l'url du service web à invoquer.

La méthode setOperationName() permet de préciser le nom de l'opération à invoquer.

La méthode invoke() permet de réaliser l'invocation du service web proprement dite.

Pour faciliter cette mise en oeuvre, Axis fournit l'outil wsdl2java qui génère des classes et interfaces à partir du wsdl du service qui sera à invoquer. Ces classes implémentent un proxy qui facilite l'invocation du service web.

Exemple : code client mettant en oeuvre le proxy généré

```
package com.jmdoudoux.test.axis;

import java.rmi.RemoteException;

import javax.xml.rpc.ServiceException;
```

```
public class TestCalculerGenere {

  public static void main(String[] args) {
    CalculerServiceLocator locator = new CalculerServiceLocator();
    long resultat;
    Calculer service;

    try {
      service = locator.getCalculer();
      resultat = service.additionner(10, 20);
      System.out.println("resultat = " + resultat);
    } catch (ServiceException e) {
      e.printStackTrace();
    } catch (RemoteException e) {
      e.printStackTrace();
    }

  }
}
```

Résultat :

```
26 déc. 2006 11:22:32 org.apache.axis.utils.JavaUtils isAttachmentSupported
ATTENTION: Unable to find required classes (javax.activation.DataHandler
and javax.mail.internet.MimeMultipart). Attachment support is disabled.
resultat = 30
```

Le proxy généré encapsule toute la mécanique d'appel. Un objet de type ServiceLocator facilite l'obtention du endpoint. L'utilisation du proxy rend le code plus simple, plus compréhensible et plus évolutif puisqu'il est généré.

54.6.1.4. L'outil TCPMonitor

Cet outil agit comme un proxy qui permet de visualiser les requêtes http échangées entre un client et un serveur.

Résultat :

```
C:\java\axis-1_4\lib>java -cp ./axis.jar org.apache.axis.utils.tcpmon 1234 localhost 8080
```

Les paramètres optionnels pouvant être fournis sont :

- le port écouté sur le client
- le hostname du serveur
- le port du serveur

Si aucun paramètre n'est fourni, l'outil affiche une première fenêtre qui permet de saisir les informations requises.

Toutes les requêtes sont faites sur un port local sur lequel l'outil écoute pour lui permettre de les afficher puis les requêtes sont envoyées au serveur. Les réponses suivent le chemin inverse pour permettre aussi leur affichage.

Cet outil est pratique pour afficher le contenu des requêtes et réponses http échangées lors des invocations.

54.6.2. Apache Axis 2

Axis 2 est le successeur du projet Axis : le projet a été complètement réécrit pour proposer une architecture plus modulaire.

Il propose un modèle de déploiement spécifique : les services web peuvent être packagés dans un fichier ayant l'extension.aar (Axis ARchive) ou contenus dans un sous répertoire du répertoire WEB-INF/services. La configuration se fait dans le fichier METE-INF/services.xml

Le runtime d'Axis 2 est une application web qui peut être utilisé dans n'importe quel serveur d'applications Java EE et même un conteneur web comme Apache Tomcat.

Des modules complémentaires permettent d'enrichir le moteur en fonctionnalités notamment le support de certaines spécifications WS-*. Chaque module est packagé dans un fichier avec l'extension .mar

Axis 2 permet de choix le framework de binding XML/Objets.

54.6.3. Xfire

XFire est un projet open source initié par la communauté CodeHaus

L'url du projet est http://xfire.codehaus.org.

Ce projet n'est plus maintenu car il a été repris par le projet CXF d'Apache.

54.6.4. Apache CXF

Apache CXF est né de la fusion des projets XFire et Celtix.

L'url du projet est **http://cxf.apache.org/**

CXF propose un support de plusieurs standards des services web notamment, SOAP 1.1 et 1.2, WSDL 1.1 et 1.2, le WS-I Basic Profile, MTOM, WS-Addressing, WS-Policy, WS-ReliableMessaging, et WS-Security.

CXF propose une api propriétaire mais implémente aussi les spécifications de JAX-WS.

CXF propose plus qu'une implémentation d'un moteur SOAP en proposant un framework complet pour le développement de services

Ses principaux objectifs sont la facilité d'utilisation, les performances, l'extensibilité et l'intégration dans d'autres systèmes. CXF utilise le framework Spring.

CXF est utilisé dans d'autres projets notamment ServiceMix et Mule.

54.6.5. JWSDP (Java Web Service Developer Pack)

Le Java Web Services Developer Pack (JWSDP) est un ensemble d'outils et d'API fournis par Sun qui permet de faciliter le développement, le déploiement et le test des services web et des applications web avec Java.

Le JWSDP contient les outils suivants :

- Apache Tomcat
- Java WSDP Registry Server (serveur UDDI)
- Web application development tool
- Apache Ant
- wscompile, wsdeploy,
- ...

La plupart de ces éléments peuvent être installés manuellement séparément. Le JWSDP propose un pack qui les regroupe en une seule installation et propose en plus des outils spécifiquement dédiés au développement de services web.

Le JWSDP contient les API particulières suivantes :

- Java XML Pack : Java API for XML Processing (JAXP), Java API for XML-based RPC (JAX-RPC), Java API for XML Messaging (JAXM), Java API for XML Registries (JAXR)
- Java Architecture for XML Binbing (JAXB)
- Java Secure Socket (JSSE)
- SOAP with Attachments API for Java (SAAJ)

JWSDP fournit aussi toutes les APIs nécessaires aux développements d'applications Web notamment les API Servlet/JSP, JSTL et JSF.

Il est possible de le télécharger sur le site de Sun : **http://java.sun.com/webservices/**.

Remarque : le projet GlassFish remplace le JWSDP.

54.6.5.1. L'installation du JWSDP 1.1

Pour pouvoir l'utiliser, il faut au minimum un jdk 1.3.1.

Il faut télécharger sur le site de Sun le fichier jwsdp-1_1-windows-i586.exe et l'exécuter.

Un assistant guide l'installation :

- Cliquer sur "Suivant".
- Lire le contrat de licence, sélectionner "Approve" et cliquer sur "Suivant".
- Sélectionner le JDK à utiliser et cliquer sur "Suivant"
- Dans le cas de l'utilisation d'un proxy, il faut renseigner les informations le concernant. Cliquer sur "Suivant".
- Sélectionner le répertoire d'installation et cliquer sur "Suivant".
- Sélectionner le type d'installation et cliquer sur "Suivant".
- Il faut saisir un nom d'utilisation qui sera l'administrateur et son mot de passe et cliquer sur "Suivant".
- L'assistant affiche un récapitulatif des options choisies. Cliquer sur "Suivant".
- Cliquer sur "Suivant".
- Cliquer sur "Suivant".
- Cliquer sur "Fin".

54.6.5.2. L'exécution du serveur

L'installation a créé une entrée dans le menu "Démarrer/Programmes".

Pour lancer le serveur d'application Tomcat, il faut utiliser l'option Start Tomcat

Attention, les ports 8080 et 8081 ne doivent pas être occupés par un autre serveur.

Pour accéder à la console d'administration, il faut lancer un navigateur sur l'url http://localhost:8081/admin

Si la page ne s'affiche pas, il faut aller voir dans le fichier catalina.out contenu dans le répertoire logs ou a été installé le JWSDP.

Il faut saisir le nom de l'utilisateur et le mot de passe saisis lors de l'installation de JWSDP.

Cette console permet de modifier les paramètres du JWSDP.

54.6.5.3. L'exécution d'un des exemples

Il faut créer un fichier build.properties dans le répertoire home (c:\document and settings\user_name) qui contient :

```
username=
password=
```

Il faut s'assurer que le chemin C:\java\jwsdp-1_0_01\bin est en premier dans le classpath surtout si une autre version de Ant est déjà installée sur la machine

Il faut lancer Tomcat puis suivre les étapes proposées ci dessous :

Résultat :

```
C:\java\jwsdp-1_0_01\docs\tutorial\examples\jaxrpc\hello>dir
Le volume dans le lecteur C s'appelle SYSTEM
Le numéro de série du volume est 18AE-3A71
Répertoire de C:\java\jwsdp-1_0_01\docs\tutorial\examples\jaxrpc\hello
03/01/2003  13:37       <DIR>          .
03/01/2003  13:37       <DIR>          ..
01/08/2002  14:16                  309 build.properties
01/08/2002  14:17                  496 build.xml
01/08/2002  14:17                  222 config.xml
01/08/2002  14:16                2 342 HelloClient.java
01/08/2002  14:17                1 999 HelloIF.java
01/08/2002  14:16                1 995 HelloImpl.java
01/08/2002  14:17                  545 jaxrpc-ri.xml
01/08/2002  14:17                  421 web.xml
               8 fichier(s)           8 329 octets
               2 Rép(s)     490 983 424 octets libres
C:\java\jwsdp-1_0_01\docs\tutorial\examples\jaxrpc\hello>ant compile-server
Buildfile: build.xml
prepare:
     [echo] Creating the required directories....
    [mkdir] Created dir: C:\java\jwsdp-1_0_01\docs\tutorial\examples\jaxrpc\hell
o\build\client\hello
    [mkdir] Created dir: C:\java\jwsdp-1_0_01\docs\tutorial\examples\jaxrpc\hell
o\build\server\hello
    [mkdir] Created dir: C:\java\jwsdp-1_0_01\docs\tutorial\examples\jaxrpc\hell
o\build\shared\hello
    [mkdir] Created dir: C:\java\jwsdp-1_0_01\docs\tutorial\examples\jaxrpc\hell
o\build\wsdeploy-generated
    [mkdir] Created dir: C:\java\jwsdp-1_0_01\docs\tutorial\examples\jaxrpc\hell
o\dist
    [mkdir] Created dir: C:\java\jwsdp-1_0_01\docs\tutorial\examples\jaxrpc\hell
o\build\WEB-INF\classes\hello
compile-server:
     [echo] Compiling the server-side source code....
    [javac] Compiling 2 source files to C:\java\jwsdp-1_0_01\docs\tutorial\examp
les\jaxrpc\hello\build\shared
BUILD SUCCESSFUL
Total time: 7 seconds
C:\java\jwsdp-1_0_01\docs\tutorial\examples\jaxrpc\hello>ant setup-web-inf
Buildfile: build.xml
setup-web-inf:
     [echo] Setting up build/WEB-INF....
   [delete] Deleting directory C:\java\jwsdp-1_0_01\docs\tutorial\examples\jaxrp
c\hello\build\WEB-INF
     [copy] Copying 2 files to C:\java\jwsdp-1_0_01\docs\tutorial\examples\jaxrp
c\hello\build\WEB-INF\classes\hello
     [copy] Copying 1 file to C:\java\jwsdp-1_0_01\docs\tutorial\examples\jaxrpc
\hello\build\WEB-INF
     [copy] Copying 1 file to C:\java\jwsdp-1_0_01\docs\tutorial\examples\jaxrpc
\hello\build\WEB-INF
BUILD SUCCESSFUL
Total time: 2 seconds
C:\java\jwsdp-1_0_01\docs\tutorial\examples\jaxrpc\hello>ant package
Buildfile: build.xml
package:
     [echo] Packaging the WAR....
      [jar] Building jar: C:\java\jwsdp-1_0_01\docs\tutorial\examples\jaxrpc\hel
lo\dist\hello-portable.war
BUILD SUCCESSFUL
Total time: 2 seconds
C:\java\jwsdp-1_0_01\docs\tutorial\examples\jaxrpc\hello>ant process-war
Buildfile: build.xml
set-ws-scripts:
process-war:
     [echo] Running wsdeploy....
     [exec] info: created temporary directory: C:\java\jwsdp-1_0_01\docs\tutoria
```

```
l\examples\jaxrpc\hello\build\wsdeploy-generated\jaxrpc-deploy-b5e49c
     [exec] info: processing endpoint: MyHello
     [exec] Note: sun.tools.javac.Main has been deprecated.
     [exec] 1 warning
     [exec] info: created output war file: C:\java\jwsdp-1_0_01\docs\tutorial\ex
amples\jaxrpc\hello\dist\hello-jaxrpc.war
     [exec] info: removed temporary directory: C:\java\jwsdp-1_0_01\docs\tutoria
l\examples\jaxrpc\hello\build\wsdeploy-generated\jaxrpc-deploy-b5e49c
BUILD SUCCESSFUL
Total time: 15 seconds
C:\java\jwsdp-1_0_01\docs\tutorial\examples\jaxrpc\hello>ant deploy
Buildfile: build.xml
deploy:
   [deploy] OK - Installed application at context path /hello-jaxrpc
   [deploy]
BUILD SUCCESSFUL
Total time: 7 seconds
C:\java\jwsdp-1_0_01\docs\tutorial\examples\jaxrpc\hello>ant generate-stubs
Buildfile: build.xml
set-ws-scripts:
prepare:
     [echo] Creating the required directories....
generate-stubs:
     [echo] Running wscompile....
     [exec] Note: sun.tools.javac.Main has been deprecated.
     [exec] 1 warning
BUILD SUCCESSFUL
Total time: 14 seconds
C:\java\jwsdp-1_0_01\docs\tutorial\examples\jaxrpc\hello>ant compile-client
Buildfile: build.xml
prepare:
     [echo] Creating the required directories....
compile-client:
     [echo] Compiling the client source code....
    [javac] Compiling 1 source file to C:\java\jwsdp-1_0_01\docs\tutorial\exampl
es\jaxrpc\hello\build\client
BUILD SUCCESSFUL
Total time: 4 seconds
C:\java\jwsdp-1_0_01\docs\tutorial\examples\jaxrpc\hello>ant jar-client
Buildfile: build.xml
jar-client:
     [echo] Building the client JAR  file....
      [jar] Building jar: C:\java\jwsdp-1_0_01\docs\tutorial\examples\jaxrpc\hel
lo\dist\hello-client.jar
BUILD SUCCESSFUL
Total time: 2 seconds
C:\java\jwsdp-1_0_01\docs\tutorial\examples\jaxrpc\hello>ant run
Buildfile: build.xml
run:
     [echo] Running the hello.HelloClient program....
     [java] Hello Duke!
BUILD SUCCESSFUL
Total time: 5 seconds

C:\java\jwsdp-1_0_01\docs\tutorial\examples\jaxrpc\hello>
```

54.6.6. Java EE 5

Java EE 5 utilise une nouvelle API pour le développement de services web : JAX-WS (Java API for XML Web Services).

54.6.7. Java SE 6

Java SE 6 fournit en standard une implémentation de JAX-WS 2.0 permettant ainsi de consommer mais aussi de produire des services web uniquement avec la plate-forme SE.

L'écriture et le déploiement d'un service web suit plusieurs étapes.

Il faut écrire la classe du service web en utilisant les annotations de JAX-WS.

Exemple :

```
package com.jmdoudoux.test.ws;

import javax.jws.WebService;

@WebService
public class TestWS {

        public String Saluer(final String nom) {
               return "Bonjour " + nom;
        }

}
```

La classe javax.xml.ws.Endpoint encapsule le endpoint d'un service web permettant ainsi son accès.

La méthode publish() permet de publier un endpoint associé à l'url fournie en paramètre.

Exemple :

```
package com.jmdoudoux.test.ws;

import javax.xml.ws.Endpoint;

public class Main {

        public static void main(String[] args) {

               System.out.println("Lancement du serveur web");

               Endpoint.publish(
                               "http://localhost:8080/ws/TestWS",
                               new TestWS());

        }

}
```

Il faut compiler la classe.

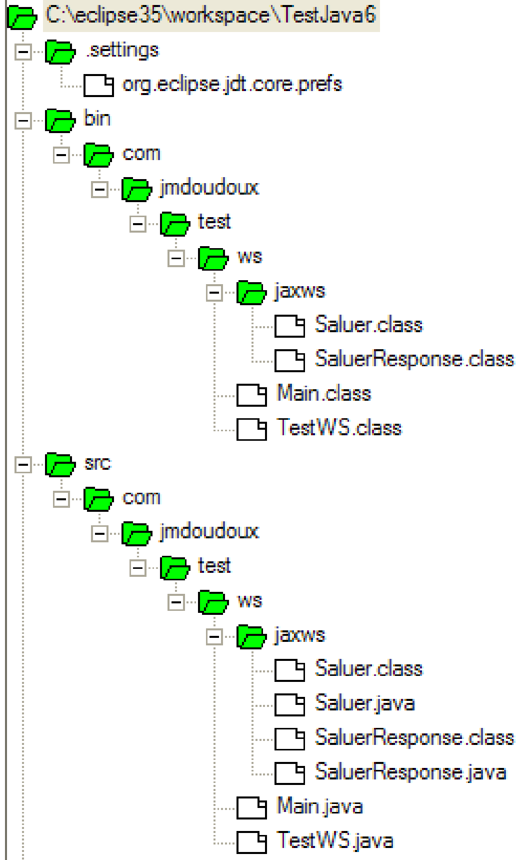
Il faut ensuite utiliser l'outil wsgen pour générer les classes JAXB qui vont mapper les requêtes et réponses des messages Soap.

Résultat :

```
C:\eclipse35\workspace\TestJava6\bin>wsgen -cp . -d ../src com.jmdoudoux.test.ws.TestWS
```

Dans l'exemple, deux classes sont générées dans le package com.jmdoudoux.test.ws.jaxws :

- Saluer : pour encapsuler la requête
- SaluerResponse : pour encapsuler la réponse



Il faut alors exécuter la classe Main : un serveur web minimaliste est lancé et le service web y est déployé.

Attention, l'environnement d'exécution doit être un JDK.

Il suffit alors d'ouvrir l'url http://localhost:8080/ws/TestWS?wsdl dans un navigateur

Le navigateur affiche alors le contenu du wsdl qui décrit le service web.

Le service web peut alors être consommé par un client, tant que l'application est en cours d'exécution.

Exemple : Le message Soap de la requête

```
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ws="http://ws.test.jmdoudoux.com/">
   <soapenv:Header/>
   <soapenv:Body>
      <ws:Saluer>
         <arg0>JM</arg0>
      </ws:Saluer>
   </soapenv:Body>
</soapenv:Envelope>
```

Exemple : Le message Soap en réponse

```
<S:Envelope
xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Body>
      <ns2:SaluerResponse
xmlns:ns2="http://ws.test.jmdoudoux.com/">
         <return>Bonjour
JM</return>
      </ns2:SaluerResponse>
   </S:Body>
</S:Envelope>
```

54.6.8. Le projet Metro et WSIT

Le projet Metro est une pile pour services web utilisée dans le serveur d'applications GlassFish V2 et V3.

Metro est l'implémentation de référence de JAX-WS.

Metro est livré avec GlassFish mais il est possible de l'utiliser dans d'autres serveurs d'applications ou conteneur web, par exemple Tomcat. Dans ce dernier cas, il faut ajouter les bibliothèques de Metro et JAXB.

Metro est composé de deux éléments :

- Une implémentation de JAX-WS pour le support des services web
- Le projet Tango qui est une implémentation de certaines spécifications WS-*

Le projet Tango est une implémentation open source des spécifications Reliability, Security et Transaction des spécifications WS-*, ce qui facilite l'interopérabilité avec le framework WCF (Windows Communication Foundation) de Microsoft .Net version 3.0 et ultérieure.

WSIT (Web Service Interoperability) est un projet commun entre Sun et Microsoft pour garantir l'intéropérabilité des piles de services web des plate-formes Java et .Net (avec le Windows Communication Framework).

Cette interopérabilité est assurée car Metro et WCF supportent tous les deux plusieurs spécifications WS-* :

- WS-Addressing
- WS-Policy
- WS-Security
- WS-Transaction
- WS-Reliable Messaging
- WS-Trust
- WS-SecureConversation

La mise en oeuvre de ces spécifications via WSIT repose sur une configuration dans un fichier XML. Le contenu de ce fichier peut être fastidieu à créer ou à modifier : Netbeans propose des assistants graphiques qui facilitent grandement leur mise en oeuvre.

54.7. Inclure des pièces jointes dans SOAP

Pour inclure des données binaires importantes dans un message SOAP, il faut utiliser le mécanisme des pièces jointes (attachment).

Malheureusement, ce mécanisme est implémenté par plusieurs standards :

- SOAP With Attachments : définis par le W3C dans la version 1.1 de SOAP
- XOP/MTOM : définis par le W3C dans la version 1.2 de SOAP

MTOM devient le standard utilisé par Java (JAX-WS) et .Net (WSE 3.0)

54.8. WS-I

Les nombreuses spécifications concernant les services web sont fréquemment incomplètes ou peu claires : il en résulte de nombreuses incompatibilités lors de leur mise en oeuvre.

Le consortium WS-I (Web Service Interoperability) **http://www.ws-i.org/** a été créé pour définir des profiles qui sont des recommandations pour permettre de faciliter l'interopérabilité des services web entre plateformes, systèmes d'exploitation et langages et pour promouvoir ces normes.

Le WS-I a définit plusieurs spécifications :

- WS-I Basic Profile
- WS-I Basic Security Profile
- Simple Soap Binding Profile
- ...

Le site web est à l'url : www.ws-i.org

54.8.1. WS-I Basic Profile

WS-I Basic Profile est un ensemble de recommandations dont le but est d'améliorer l'interopérabilité entre les différents moteurs SOAP.

La suite de cette section sera développée dans une version future de ce document

54.9. Les autres spécifications

Les spécifications SOAP et WSDL permettent de réaliser des échanges de messages basiques. L'accroissement de l'utilisation des services web a fait émerger la nécessité de fonctionnalités supplémentaires telles que la gestion de la sécurité, des transactions, de la fiabilité des messages, ...

Les spécifications désignées sous l'acronyme WS-* concernent les spécifications de seconde génération des services web (elles étendent les spécifications de la première génération de spécifications constituée par SOAP, WSDL, UDDI). L'abréviation WS-* est communément utilisée car la majorité de ces spécifications commence par WS-

De nombreuses autres spécifications sont en cours d'élaboration et de tentatives de standardisations ou de reconnaissance par le marché.

Fréquemment ces spécifications sont complémentaires ou dépendantes voir même dans quelques cas concurrentes car elles sont soutenues par des acteurs du marché ou des organismes de standardisation différents. Il est généralement nécessaire d'utiliser plusieurs de ces spécifications pour permettre de répondre aux besoins notamment en terme de sécurité, fiabilité, ...

Il est aussi très important de tenir compte de la maturité d'une spécification avant de la mettre en oeuvre.

Ces spécifications permettent de mettre en oeuvre des scénarios complexes impliquant l'utilisation de services web.

Toutes ces spécifications requièrent l'utilisation de SOAP.

La suite de cette section sera développée dans une version future de ce document

Partie 8 : Le développement d'applications web

Cette partie contient plusieurs chapitres :

- Les servlets : plonge au coeur de l'API servlet qui est un des composants de base pour le développement d'applications Web
- Les JSP (Java Server Pages) : poursuit la discussion avec les servlets en explorant un mécanisme basé sur celles ci pour réaliser facilement des pages web dynamiques
- JSTL (Java server page Standard Tag Library) : est un ensemble de bibliothèques de tags personnalisés communément utilisé dans les JSP
- Struts : présente et détaille la mise en oeuvre de ce framework open source de développement d'applications web le plus populaire
- JSF (Java Server Faces) : détaille l'utilisation de la technologie Java Server Faces (JSF) dont le but est de proposer un framework qui facilite et standardise le développement d'applications web avec Java.
- D'autres frameworks pour les applications web : présente rapidement quelques frameworks open source pour le développement d'applications web

55. Les servlets

Les serveurs web sont de base uniquement capables de renvoyer des fichiers présents sur le serveur en réponse à une requête d'un client. Cependant, pour permettre l'envoi d'une page HTML contenant par exemple une liste d'articles répondant à différents critères, il faut créer dynamiquement ces pages HTML. Plusieurs solutions existent pour ces traitements. Les servlets java sont une des ces solutions.

Mais les servlets peuvent aussi servir à d'autres usages.

Sun fourni des informations sur les servlets sur son site : http://java.sun.com/products/servlet/index.html

Ce chapitre contient plusieurs sections :

- La présentation des servlets
- L'API servlet
- Le protocole HTTP
- Les servlets http
- Les informations sur l'environnement d'exécution des servlets
- L'utilisation des cookies
- Le partage d'informations entre plusieurs échanges HTTP
- Packager une application web
- L'utilisation Log4J dans une servlet

55.1. La présentation des servlets

Une servlet est un programme qui s'exécute côté serveur en tant qu'extension du serveur. Elle reçoit une requête du client, elle effectue des traitements et renvoie le résultat. La liaison entre la servlet et le client peut être directe ou passer par un intermédiaire comme par exemple un serveur http.

Même si pour le moment la principale utilisation des servlets est la génération de pages html dynamiques utilisant le protocole http et donc un serveur web, n'importe quel protocole reposant sur le principe de requête/réponse peut faire usage d'une servlet.

Ecrite en java, une servlet en retire ses avantages : la portabilité, l'accès à toutes les API de java dont JDBC pour l'accès aux bases de données, ...

Une servlet peut être invoquée plusieurs fois en même temps pour répondre à plusieurs requêtes simultanées.

La servlet se positionne dans une architecture Client/Serveur trois tiers dans le tiers du milieu entre le client léger chargé de l'affichage et la source de données.

Il existe plusieurs versions des spécifications de l'API Servlets :

Version	
2.0	1997
2.1	

	Novembre 1998, partage d'informations grâce au Servletcontext La classe GenericServlet implémente l'interface ServletConfig une méthode log() standard pour envoyer des informations dans le journal du conteneur objet RequestDispatcher pour le transfert du traitement de la requête vers une autre ressource ou inclure le résultat d'une autre ressource
2.2	Aout 1999, format war pour un déploiement standard des applications web mise en buffer de la réponse inclus dans J2EE 1.2
2.3	Septembre 2001, JSR 053 : nécessite le JDK 1.2 minimum ajout d'un mécanisme de filtre ajout de méthodes pour la gestion d'événements liés à la création et la destruction du context et de la session inclus dans J2EE 1.3
2.4	Novembre 2003, JSR 154 inclus dans J2EE 1.4

55.1.1. Le fonctionnement d'une servlet (cas d'utilisation de http)

Un serveur d'application permet de charger et d'exécuter les servlets dans une JVM. C'est une extension du serveur web. Ce serveur d'application contient entre autre un moteur de servlets qui se charge de manager les servlets qu'il contient.

Pour exécuter une servlet, il suffit de saisir une URL qui désigne la servlet dans un navigateur.

1. Le serveur reçoit la requête http qui nécessite une servlet de la part du navigateur
2. Si c'est la première sollicitation de la servlet, le serveur l'instancie. Les servlets sont stockées (sous forme de fichier .class) dans un répertoire particulier du serveur. Ce répertoire dépend du serveur d'application utilisé. La servlet reste en mémoire jusqu'à l'arrêt du serveur. Certains serveurs d'application permettent aussi d'instancier des servlets dès le lancement du serveur.

   La servlet en mémoire, peut être appelée par plusieurs threads lancés par le serveur pour chaque requête. Ce principe de fonctionnement évite d'instancier un objet de type servlet à chaque requête et permet de maintenir un ensemble de ressources actives tel qu'une connexion à une base de données.
3. le serveur créé un objet qui représente la requête http et objet qui contiendra la réponse et les envoie à la servlet
4. la servlet créé dynamiquement la réponse sous forme de page html transmise via un flux dans l'objet contenant la réponse. La création de cette réponse utilise bien sûre la requête du client mais aussi un ensemble de ressources incluses sur le serveur tels de que des fichiers ou des bases de données.
5. le serveur récupère l'objet réponse et envoie la page html au client.

55.1.2. Les outils nécessaires pour développer des servlets

Initialement, pour développer des servlets avec le JDK standard édition, il faut utiliser le Java Server Development Kit (JSDK) qui est une extension du JDK. Pour réaliser les tests, le JSDK fournit, dans sa version 2.0 un outil nommé servletrunner et depuis sa version 2.1, il fournit un serveur http allégé.

Actuellement, pour exécuter des applications web, il faut utiliser un conteneur web ou serveur d'application : il existe de nombreuses versions commerciales tel que IBM Webpshere ou BEA WebLogic mais aussi des versions libres tel que Tomcat du projet GNU Jakarta.

Ce serveur d'application ou ce conteneur web doit utiliser ou inclure un serveur http dont le plus utilisé est Apache.

Le choix d'un serveur d'application ou d'un conteneur web doit tenir compte de la version du JSDK qu'il supporte pour être compatible avec celle utilisée pour le développement des servlets. Le choix entre un serveur commercial et un libre doit tenir compte principalement du support technique, des produits annexes fournis et des outils d'installation et de configuration.

Pour simplement développer des servlets, le choix d'un serveur libre se justifie pleinement de part leur gratuité et leur « légèreté ».

55.1.3. Le rôle du conteneur web

Un conteneur web est un moteur de servlet qui prend en charge et gère les servlets : chargement de la servlet, gestion de son cycle de vie, passage des requêtes et des réponses ... Un conteur web peut être intégré dans un serveur d'application qui va contenir d'autre conteneur et éventuellement proposer d'autres services..

Le chargement et l'instanciation d'une servlet se font selon le paramétrage soit au lancement du serveur soit à la première invocation de la servlet. Dès l'instanciation, la servlet est initialisée une seule et unique fois avant de pouvoir répondre aux requêtes. Cette initialisation peut permettre de mettre en place l'accès à des ressources tel qu'une base de données.

55.1.4. Les différences entre les servlets et les CGI

Les programmes ou script CGI (Common Gateway Interface) sont aussi utilisés pour générer des pages HTML dynamiques. Ils représentent la plus ancienne solution pour réaliser cette tache.

Un CGI peut être écrit dans de nombreux langages.

Il existe plusieurs avantages à utiliser des servlets plutôt que des CGI :

- la portabilité offerte par Java bien que certains langages de script tel que PERL tourne sur plusieurs plateformes.
- la servlet reste en mémoire une fois instanciée ce qui permet de garder des ressources systèmes et gagner le temps de l'initialisation. Un CGI est chargé en mémoire à chaque requête, ce qui réduit les performances.
- les servlets possèdent les avantages de toutes les classes écrites en java : accès aux API, aux java beans, le garbage collector, ...

55.2. L'API servlet

Les servlets sont conçues pour agir selon un modèle de requête/réponse. Tous les protocoles utilisant ce modèle peuvent être utilisés tel que http, ftp, etc ...

L'API servlets est une extension du jdk de base, et en tant que telle elle est regroupée dans des packages préfixés par javax

L'API servlet regroupe un ensemble de classes dans deux packages :

- javax.servlet : contient les classes pour développer des serlvets génériques indépendantes d'un protocole
- javax.servlet.http : contient les classes pour développer des servlets qui reposent sur le protocole http utilisé par les serveurs web.

Le package javax.servlet définit plusieurs interfaces, méthodes et exceptions :

javax.servlet	Nom	Rôle
Les interfaces	RequestDispatcher	Définition d'un objet qui permet le renvoi d'une requête vers une autre ressource du serveur (une autre servlet, une JSP ...)
	Servlet	Définition de base d'une servlet
	ServletConfig	Définition d'un objet pour configurer la servlet
	ServletContext	Définition d'un objet pour obtenir des informations sur le contexte d'exécution de la servlet

	ServletRequest	Définition d'un objet contenant la requête du client
	ServletResponse	Définition d'un objet qui contient la réponse renvoyée par la servlet
	SingleThreadModel	Permet de définir une servlet qui ne répondra qu'à une seule requête à la fois
Les classes	GenericServlet	Classe définissant une servlet indépendante de tout protocole
	ServletInputStream	Flux permet la lecture des donnes de la requête cliente
	ServletOutPutStream	Flux permettant l'envoie de la réponse de la servlet
Les exceptions	SevletException	Exception générale en cas de problème durant l'exécution de la servlet
	UnavailableException	Exception levée si la servlet n'est pas disponible

Le package javax.servlet.http définit plusieurs interfaces et méthodes :

Javax.servlet	Nom	Rôle
Les interfaces	HttpServletRequest	Hérite de ServletRequest : définit un objet contenant une requête selon le protocole http
	HttpServletResponse	Hérite de ServletResponse : définit un objet contenant la réponse de la servlet selon le protocole http
	HttpSession	Définit un objet qui représente une session
Les classes	Cookie	Classe représentant un cookie (ensemble de données sauvegardées par le brower sur le poste client)
	HttpServlet	Hérite de GenericServlet : classe définissant une servlet utilisant le protocole http
	HttpUtils	Classe proposant des méthodes statiques utiles pour le développement de servlet http

55.2.1. L'interface Servlet

Une servlet est une classe Java qui implémente l'interface javax.servlet.Servlet. Cette interface définit 5 méthodes qui permettent au conteneur web de dialoguer avec la servlet : elle encapsule ainsi les méthodes nécessaires à la communication entre le conteneur et la servlet.

Méthode	Rôle
void service (ServletRequest req, ServletResponse res)	Cette méthode est exécutée par le conteneur lorsque la servlet est sollicitée : chaque requête du client déclenche une seule exécution de cette méthode. Cette méthode pouvant être exécutée par plusieurs threads, il faut prévoir un processus d'exclusion pour l'utilisation de certaines ressources.
void init(ServletConfig conf)	Initialisation de la servlet. Cette méthode est appelée une seule fois après l'instanciation de la servlet. Aucun traitement ne peut être effectué par la servlet tant que l'exécution de cette méthode n'est pas terminée.
ServletConfig getServletConfig()	Renvoie l'objet ServletConfig passé à la méthode init
void destroy()	Cette méthode est appelée lors de la destruction de la servlet. Elle permet de libérer proprement certaines ressources (fichiers, bases de données ...). C'est le serveur qui appelle cette méthode.

String getServletInfo()	Renvoie des informations sur la servlet.

Les méthodes init(), service() et destroy() assurent le cycle de vie de la servlet en étant respectivement appelées lors de la création de la servlet, lors de son appel pour le traitement d'une requête et lors de sa destruction.

La méthode init() est appelée par le serveur juste après l'instanciation de la servlet.

La méthode service() ne peut pas être invoquée tant que la méthode init() n'est pas terminée.

La méthode destroy() est appelée juste avant que le serveur ne détruise la servlet : cela permet de libérer des ressources allouées dans la méthode init() tel qu'un fichier ou une connexion à une base de données.

55.2.2. La requête et la réponse

L'interface ServletRequest définit plusieurs méthodes qui permettent d'obtenir des données sur la requête du client :

Méthode	Rôle
ServletInputStream getInputStream()	Permet d'obtenir un flux pour les données de la requête
BufferedReader getReader()	Idem

L'interface ServletResponse définit plusieurs méthodes qui permettent de fournir la réponse faite par la servlet suite à ces traitements :

Méthode	Rôle
SetContentType	Permet de préciser le type MIME de la réponse
ServletOutputStream getOutputStream()	Permet d'obtenir un flux pour envoyer la réponse
PrintWriter getWriter()	Permet d'obtenir un flux pour envoyer la réponse

55.2.3. Un exemple de servlet

Une servlet qui implémente simplement l'interface Servlet doit évidemment redéfinir toutes les méthodes définies dans l'interface.

Il est très utile lorsque que l'on crée une servlet qui implémente directement l'interface Servlet de sauvegarder l'objet ServletConfig fourni par le conteneur en paramètre de la méthode init() car c'est le seul moment où l'on a accès à cet objet.

Exemple (code Java 1.1) :

```
import java.io.*;
import javax.servlet.*;

public class TestServlet implements Servlet {
  private ServletConfig cfg;

  public void init(ServletConfig config) throws ServletException {
     cfg = config;
  }

  public ServletConfig getServletConfig() {
    return cfg;
  }

  public String getServletInfo() {
```

```
    return "Une servlet de test";
  }

  public void destroy() {
  }

  public void service (ServletRequest req,  ServletResponse res )
  throws ServletException, IOException  {
    res.setContentType( "text/html" );
    PrintWriter out = res.getWriter();
    out.println( "<THML>" );
    out.println( "<HEAD>");
    out.println( "<TITLE>Page generee par une servlet</TITLE>" );
    out.println( "</HEAD>" );
    out.println( "<BODY>" );
    out.println( "<H1>Bonjour</H1>" );
    out.println( "</BODY>" );
    out.println( "</HTML>" );
    out.close();
  }
}
```

55.3. Le protocole HTTP

Le protocole HTTP est un protocole qui fonctionne sur le modèle client/serveur. Un client qui est une application (souvent un navigateur web) envoie une requête à un serveur (un serveur web). Ce serveur attend en permanence les requêtes sur un port particulier (par défaut le port 80). A la réception de la requête, le serveur lance un thread qui va la traiter pour générer la réponse. Le serveur renvoie la réponse au client une fois les traitements terminés.

Une particularité du protocole HTTP est de maintenir la connexion entre le client et le serveur uniquement durant l'échange de la requête et de la réponse.

Il existe deux versions principales du protocole HTTP : 1.0 et 1.1.

La requête est composée de trois parties :

- la commande
- la section en-tête
- le corps

La première ligne de la requête contient la commande à exécuter par le serveur. La commande est suivie éventuellement d'un argument qui précise la commande (par exemple l'url de la ressource demandée). Enfin la ligne doit contenir la version du protocole HTTP utilisé, précédée de HTTP/.

Exemple :

GET / index.html HTTP/1.0

Avec HTTP 1.1, les commandes suivantes sont définies : GET, POST, HEAD, OPTIONS, PUT, DELETE, TRACE et CONNECT. Les trois premières sont les plus utilisées.

Il est possible de fournir sur les lignes suivantes de la partie en-tête des paramètres supplémentaires. Cette partie en-tête est optionnelle. Les informations fournies peuvent permettre au serveur d'obtenir des informations sur le client. Chaque information doit être mise sur une ligne unique. Le format est nom_du_champ:valeur. Les champs sont prédéfinis et sont sensibles à la casse.

Une ligne vide doit précéder le corps de la requête. Le contenu du corps de la requête dépend du type de la commande.

La requête doit obligatoirement être terminée par une ligne vide.

La réponse est elle aussi composée des trois mêmes parties :

- une ligne de statuts
- un en-tête dont le contenu est normalisé
- un corps dont le contenu dépend totalement de la requête

La première ligne de l'en-tête contient un état qui est composé : de la version du protocole HTTP utilisé, du code de statut et d'une description succincte de ce code.

Le code de statut est composé de trois chiffres qui donnent des informations sur le résultat du traitement qui a généré cette réponse. Ce code peut être regroupé en plusieurs catégories en fonction de leur valeur :

Plage de valeur du code	Signification
100 à 199	Information
200 à 299	Traitement avec succès
300 à 399	La requête a été redirigée
400 à 499	La requête est incomplète ou erronée
500 à 599	Une erreur est intervenue sur le serveur

Plusieurs codes sont définis par le protocole HTTP dont les plus importants sont :

- 200 : traitement correct de la requête
- 204 : traitement correct de la requête mais la réponse ne contient aucun contenu (ceci permet au browser de laisser la page courante affichée)
- 404 : la ressource demande n'est pas trouvée (sûrement le plus célèbre)
- 500 : erreur interne du serveur

L'en-tête contient des informations qui précisent le contenu de la réponse.

Le corps de la réponse est précédé par une ligne vide.

La suite de cette section sera développée dans une version future de ce document

55.4. Les servlets http

L'usage principal des servlets est la création de pages HTML dynamiques. Sun fourni une classe qui encapsule une servlet utilisant le protocole http. Cette classe est la classe HttpServlet.

Cette classe hérite de GenericServlet, donc elle implémente l'interface Servlet, et redéfinie toutes les méthodes nécessaires pour fournir un niveau d'abstraction permettant de développer facilement des servlets avec le protocole http.

Ce type de servlet n'est pas utile seulement pour générer des pages HTML bien que cela soit son principal usage, elle peut aussi réaliser un ensemble de traitements tel que mettre à jour une base de données. En réponse, elle peut générer une page html qui indique le succès ou non de la mise à jour. Une servlets peut aussi par exemple renvoyer une image qu'elle aura dynamiquement générée en fonction de certains paramètres.

Elle définit un ensemble de fonctionnalités très utiles : par exemple, elle contient une méthode service() qui appelle certaines méthodes à redéfinir en fonction du type de requête http (doGet(), doPost(), etc ...).

La requête du client est encapsulée dans un objet qui implémente l'interface HttpServletRequest : cet objet contient les données de la requête et des informations sur le client.

La réponse de la servlet est encapsulée dans un objet qui implémente l'interface HttpServletResponse.

Typiquement pour définir une servlet, il faut définir une classe qui hérite de la classe HttpServlet et redéfinir la classe doGet et/ou doPost selon les besoins.

La méthode service héritée de HttpServlet appelle l'une ou l'autre de ces méthodes en fonction du type de la requête http :

- une requête GET : c'est une requête qui permet au client de demander une ressource
- une requête POST : c'est une requête qui permet au client d'envoyer des informations issues par exemple d'un formulaire

Une servlet peut traiter un ou plusieurs types de requêtes grâce à plusieurs autres méthodes :

- doHead() : pour les requêtes http de type HEAD
- doPut() : pour les requêtes http de type PUT
- doDelete() : pour les requêtes http de type DELETE
- doOptions() : pour les requêtes http de type OPTIONS
- doTrace() : pour les requêtes http de type TRACE

La classe HttpServlet hérite aussi de plusieurs méthodes définies dans l'interface Servlet : init(), destroy() et getServletInfo().

55.4.1. La méthode init()

Si cette méthode doit être redéfinie, il est important d'invoquer la méthode héritée avec un appel à super.init(config), config étant l'objet fourni en paramètre de la méthode. Cette méthode définie dans la classe HttpServlet sauvegarde l'objet de type ServletConfig.

De plus, la classe GenericServlet implémente l'interface ServletConfig. Les méthodes redéfinies pour cette interface utilisent l'objet sauvegardé. Ainsi, la servlet peut utiliser sa propre méthode getInitParameter() ou utiliser la méthode getInitParameter() de l'objet de type ServletConfig. La première solution permet un usage plus facile dans toute la servlet.

Sans l'appel à la méthode héritée lors d'une redéfinition, la méthode getInitParameter() de la servlet lèvera une exception de type NullPointerException.

55.4.2. L'analyse de la requête

La méthode service() est la méthode qui est appelée lors d'un appel à la servlet.

Par défaut dans la classe HttpServlet, cette méthode contient du code qui réalise une analyse de la requête client contenue dans l'objet HttpServletRequest. Selon le type de requête GET ou POST, elle appelle la méthode doGet() ou doPost(). C'est bien ce type de requête qui indique quelle méthode utiliser dans la servlet.

Ainsi, la méthode service() n'est pas à redéfinir pour ces requêtes et il suffit de redéfinir les méthodes doGet() et/ou doPost() selon les besoins.

55.4.3. La méthode doGet()

Une requête de type GET est utile avec des liens. Par exemple :

```
<A HREF="http://localhost:8080/examples/servlet/tomcat1.MyHelloServlet">test de
la servlet</A>
```

Dans une servlet de type HttpServlet, une telle requête est associée à la méthode doGet().

La signature de la méthode doGet() :

```
protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws IOException
{

}
```

Le traitement typique de la méthode doGet() est d'analyser les paramètres de la requête, alimenter les données de l'en-tête de la réponse et d'écrire la réponse.

55.4.4. La méthode doPost()

Une requête POST n'est utilisable qu'avec un formulaire HTML.

Exemple : de code HTML

```
<FORM ACTION="http://localhost:8080/examples/servlet/tomcat1.TestPostServlet"
METHOD="POST">
<INPUT NAME="NOM">
<INPUT NAME="PRENOM">
<INPUT TYPE="ENVOYER">
</FORM>
```

Dans l'exemple ci dessus, le formulaire comporte deux zones de saisies correspondant à deux paramètres : NOM et PRENOM.

Dans une servlet de type HttpServlet, une telle requête est associée à la méthode doPost().

La signature de la méthode doPost() :

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws IOException
{

}
```

La méthode doPost() doit généralement recueillir les paramètres pour les traiter et générer la réponse. Pour obtenir la valeur associée à chaque paramètre il faut utiliser la méthode getParameter() de l'objet HttpServletRequest. Cette méthode attend en paramètre le nom du paramètre dont on veut la valeur. Ce paramètre est sensible à la casse.

Exemple :

```
public void doPost(HttpServletRequest request, HttpServletResponse response)
throws IOException, ServletException
{
   String nom = request.getParameter("NOM");
   String prenom = request.getParameter("PRENOM");
}
```

55.4.5. La génération de la réponse

La servlet envoie sa réponse au client en utilisant un objet de type HttpServetResponse. HttpServletResponse est une interface : il n'est pas possible d'instancier un tel objet mais le moteur de servlet instancie un objet qui implémente cette interface et le passe en paramètre de la méthode service.

Cette interface possède plusieurs méthodes pour mettre à jour l'en-tête http et le page HTML de retour.

Méthode	Rôle
void sendError (int)	Envoie une erreur avec un code retour et un message par défaut
void sendError (int, String)	Envoie une erreur avec un code retour et un message
void setContentType(String)	Héritée de ServletResponse, cette méthode permet de préciser le type MIME de la réponse
void setContentLength(int)	Héritée de ServletResponse, cette méthode permet de préciser la longueur de la réponse
ServletOutputStream getOutputStream()	Héritée de ServletResponse, elle retourne un flux pour l'envoi de la réponse
PrintWriter getWriter()	Héritée de ServletResponse, elle retourne un flux pour l'envoi de la réponse

Avant de générer la réponse sous forme de page HTML, il faut indiquer dans l'en-tête du message http, le type mime du contenu du message. Ce type sera souvent « text/html » qui correspond à une page HTML mais il peut aussi prendre d'autres valeurs en fonction de ce que retourne la servlet (une image par exemple). La méthode à utiliser est setContentType().

Il est aussi possible de préciser la longueur de la réponse avec la méthode setContentLength(). Cette précision est optionnelle mais si elle est utilisée, la longueur doit être exacte pour éviter des problèmes.

Il est préférable de créer une ou plusieurs méthodes recevant en paramètre l'objet HttpServletResponse qui seront dédiées à la génération du code HTML afin de ne pas alourdir les méthodes doXXX().

Il existe plusieurs façons de générer une page HTML : elles utiliseront toutes soit la méthode getOutputStream() ou getWriter() pour obtenir un flux dans lequel la réponse sera envoyée.

- Utilisation d'un StringBuffer et getOutputStream

Exemple (code Java 1.1) :

```
protected void GenererReponse1(HttpServletResponse reponse) throws IOException
{
   //creation de la reponse
   StringBuffer sb = new StringBuffer();
   sb.append("<HTML>\n");
   sb.append("<HEAD>\n");
   sb.append("<TITLE>Bonjour</TITLE>\n");
   sb.append("</HEAD>\n");
   sb.append("<BODY>\n");
   sb.append("<H1>Bonjour</H1>\n");
   sb.append("</BODY>\n");
   sb.append("</HTML>");

   // envoie des infos de l'en tete
   reponse.setContentType("text/html");
   reponse.setContentLength(sb.length());

   // envoie de la reponse
   reponse.getOutputStream().print(sb.toString());
}
```

L'avantage de cette méthode est qu'elle permet facilement de déterminer la longueur de la réponse.

Dans l'exemple, l'ajout des retours chariot '\n' à la fin de chaque ligne n'est pas obligatoire mais elle facilite la compréhension du code HTML surtout si il devient plus complexe.

* Utilisation directe de getOutputStream

Exemple :

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class TestServlet4 extends HttpServlet {

   public void doGet(HttpServletRequest req, HttpServletResponse res)
   throws ServletException, IOException {
      res.setContentType("text/html");
      ServletOutputStream out = res.getOutputStream();
      out.println("<HTML>\n");
      out.println("<HEAD>\n");
      out.println("<TITLE>Bonjour</TITLE>\n");
      out.println("</HEAD>\n");
      out.println("<BODY>\n");
      out.println("<H1>Bonjour</H1>\n");
      out.println("</BODY>\n");
      out.println("</HTML>");
   }

}
```

* Utilisation de la méthode getWriter()

Exemple (code Java 1.1) :

```
protected void GenererReponse2(HttpServletResponse reponse) throws IOException {

   reponse.setContentType("text/html");

   PrintWriter out = reponse.getWriter();

   out.println("<HTML>");
   out.println("<HEAD>");
   out.println("<TITLE>Bonjour</TITLE>");
   out.println("</HEAD>");
   out.println("<BODY>");
   out.println("<H1>Bonjour</H1>");
   out.println("</BODY>");
   out.println("</HTML>");
}
```

Avec cette méthode, il faut préciser le type MIME avant d'écrire la réponse. L'emploi de la méthode println() permet d'ajouter un retour chariot en fin de chaque ligne.

Si un problème survient lors de la génération de la réponse, la méthode sendError() permet de renvoyer une erreur au client : un code retour est positionné dans l'en-tête http et le message est indiqué dans une simple page HTML.

55.4.6. Un exemple de servlet HTTP très simple

Toute servlet doit au moins importer trois packages : java.io pour la gestion des flux et deux packages de l'API servlet ; javax.servlet.* et javax.servlet.http.

Il faut déclarer une nouvelle classe qui hérite de HttpServlet.

Il faut redéfinir la méthode doGet() pour y insérer le code qui va envoyer dans un flux le code HTML de la page générée.

Exemple (code Java 1.1) :

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class MyHelloServlet extends HttpServlet {

   public void doGet(HttpServletRequest request, HttpServletResponse response)
   throws IOException, ServletException {

      response.setContentType("text/html");

      PrintWriter out = response.getWriter();

      out.println("<html>");
      out.println("<head>");
      out.println("<title>Bonjour tout le monde</title>");
      out.println("</head>");
      out.println("<body>");
      out.println("<h1>Bonjour tout le monde</h1>");
      out.println("</body>");
      out.println("</html>");
   }
}
```

La méthode getWriter() de l'objet HttpServletResponse renvoie un flux de type PrintWriter dans lequel on peut écrire la réponse.

Si aucun traitement particulier n'est associé à une requête de type POST, il est pratique de demander dans la méthode doPost() d'exécuter la méthode doGet(). Dans ce cas, la servlet est capable de renvoyer une réponse pour les deux types de requête.

Exemple (code Java 1.1) :

```
public void doPost(HttpServletRequest request,HttpServletResponse response)
throws ServletException,IOException {

   this.doGet(request, response);

}
```

55.5. Les informations sur l'environnement d'exécution des servlets

Une servlet est exécutée dans un contexte particulier mis en place par le moteur de servlet.

La servlet peut obtenir des informations sur ce contexte.

La servlet peut aussi obtenir des informations à partir de la requête du client.

55.5.1. Les paramètres d'initialisation

Dès que de la servlet est instanciée, le moteur de servlet appelle sa méthode init() en lui donnant en paramètre un objet de type ServletConfig.

ServletConfig est une interface qui possède deux méthodes permettant de connaître les paramètres d'initialisation :

- String getInitParameter(String) : retourne la valeur du paramètre dont le nom est fourni en paramètre

Exemple :

```
String param;

public void init(ServletConfig config) {

   param = config.getInitParameter("param");

}
```

- Enumeration getInitParameterNames() : retourne une énumération des paramètres d'initialisation

Exemple (code Java 1.1) :

```
public void init(ServletConfig config) throws ServletException {

   cfg = config;

   System.out.println("Liste des parametres d'initialisation");

   for (Enumeration e=config.getInitParameterNames(); e.hasMoreElements();) {

      System.out.println(e.nextElement());

   }

}
```

La déclaration des paramètres d'initialisation dépend du serveur qui est utilisé.

55.5.2. L'objet ServletContext

La servlet peut obtenir des informations à partir d'un objet ServletContext retourné par la méthode getServletContext() d'un objet ServletConfig.

Il est important de s'assurer que cet objet ServletConfig, obtenu par la méthode init() est soit explicitement sauvegardé soit sauvegardé par l'appel à la méthode init() héritée qui effectue cette sauvegarde.

L'interface ServletContext contient plusieurs méthodes dont les principales sont :

méthode	Rôle	Deprecated
String getMimeType(String)	Retourne le type MIME du fichier en paramètre	
String getServletInfo()	Retourne le nom et le numéro de version du moteur de servlet	
Servlet getServlet(String)	Retourne une servlet à partir de son nom grâce au context	Ne plus utiliser depuis la version 2.1 du jsdk
Enumeration getServletNames()	Retourne une énumération qui contient la liste des servlets relatives au contexte	Ne plus utiliser depuis la version 2.1 du jsdk
void log(Exception, String)	Ecrit les informations fournies en paramètre dans le fichier log du serveur	Utiliser la nouvelle méthode surchargée log()
void log(String)	Idem	
void log (String, Throwable)	Idem	

Exemple : écriture dans le fichier log du serveur

```
public void init(ServletConfig config) throws ServletException {

   ServletContext sc = config.getServletContext();

   sc.log( "Demarrage servlet TestServlet" );

}
```

Le format du fichier log est dépendant du serveur utilisé :

Exemple : résultat avec tomcat

```
Context log path="/examples" :Demarrage servlet TestServlet
```

55.5.3. Les informations contenues dans la requête

De nombreuses informations en provenance du client peuvent être extraites de l'objet ServletRequest passé en paramètre par le serveur (ou de HttpServletRequest qui hérite de ServletRequest).

Les informations les plus utiles sont les paramètres envoyés dans la requête.

L'interface ServletRequest dispose de nombreuses méthodes pour obtenir ces informations :

Méthode	Rôle
int getContentLength()	Renvoie la taille de la requête, 0 si elle est inconnue
String getContentType()	Renvoie le type MIME de la requête, null si il est inconnu
ServletInputStream getInputStream()	Renvoie un flux qui contient le corps de la requête
Enumeration getParameterNames()	Renvoie une énumération contenant le nom de tous les paramètres
String getProtocol()	Retourne le nom du protocole et sa version utilisé par la requête
BufferedReader getReader()	Renvoie un flux qui contient le corps de la requête
String getRemoteAddr()	Renvoie l'adresse IP du client
String getRemoteHost()	Renvoie le nom de la machine cliente
String getScheme	Renvoie le protocole utilisé par la requête (exemple : http, ftp ...)
String getServerName()	Renvoie le nom du serveur qui a reçu la requête
int getServerPort()	Renvoie le port du serveur qui a reçu la requête

Exemple (code Java 1.1) :

```
package tomcat1;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

public class InfoServlet extends HttpServlet {

   public void doGet(HttpServletRequest request, HttpServletResponse response)
   throws IOException,ServletException {

      GenererReponse(request, response);

   }
```

```
    protected void GenererReponse(HttpServletRequest request, HttpServletResponse reponse)
    throws IOException {

       reponse.setContentType("text/html");

       PrintWriter out =reponse.getWriter();

       out.println("<html>");
       out.println("<body>");
       out.println("<head>");
       out.println("<title>Informationsa disposition de la servlet</title>");
       out.println("</head>");
       out.println("<body>");
       out.println("<p>Typemime de la requête :"
         +request.getContentType()+"</p>");
       out.println("<p>Protocolede la requête :"
         +request.getProtocol()+"</p>");
       out.println("<p>AdresseIP du client :"
         +request.getRemoteAddr()+"</p>");
       out.println("<p>Nom duclient : "
         +request.getRemoteHost()+"</p>");
       out.println("<p>Nom duserveur qui a reçu la requête :"
         +request.getServerName()+"</p>");
       out.println("<p>Port duserveur qui a reçu la requête :"
         +request.getServerPort()+"</p>");
       out.println("<p>scheme: "+request.getScheme()+"</p>");
       out.println("<p>listedes parametres </p>");

       for (Enumeration e =request.getParameterNames() ; e.hasMoreElements() ; ) {

          Object p = e.nextElement();
          out.println("<p>  nom : "+p+" valeur :"
             +request.getParameter(""+p)+"</p>");

       }

       out.println("</body>");
       out.println("</html>");

    }
}
```

Résultat : avec l'url http://localhost:8080/examples/servlet/tomcat1.InfoServlet?param1=valeur1¶m2=valeur2 : Une page html s'affiche contenant :

```
Type mime de la requête : null

Protocole de la requête : HTTP/1.0

Adresse IP du client : 127.0.0.1

Nom du client : localhost

Nom du serveur qui a reçu la requête : localhost

Port du serveur qui a reçu la requête : 8080

scheme : http

liste des parametres

   nom : param2 valeur :valeur2

   nom : param1 valeur :valeur1
```

55.6. L'utilisation des cookies

Les cookies sont des fichiers contenant des données au format texte, envoyés par le serveur et stockés sur le poste client. Les données contenues dans le cookie sont renvoyées au serveur à chaque requête.

Les cookies peuvent être utilisés explicitement ou implicitement par exemple lors de l'utilisation d'une session.

Les cookies ne sont pas dangereux car ce sont uniquement des fichiers textes qui ne sont pas exécutés. De plus, les navigateurs posent des limites sur le nombre (en principe 20 cookies pour un même serveur) et la taille des cookies (4ko maximum). Par contre les cookies peuvent contenir des données plus ou moins sensibles. Il est capital de ne stocker dans les cookies que des données qui ne sont pas facilement exploitables par une intervention humaine sur le poste client et tout cas de ne jamais les utiliser pour stocker des informations sensibles tel qu'un numéro de carte bleue.

55.6.1. La classe Cookie

La classe javax.servlet.http.Cookie encapsule un cookie.

Un cookie est composé d'un nom, d'une valeur et d'attributs.

Pour créer un cookie, il suffit d'instancier un nouvel objet de type Cookie. La classe Cookie ne possède qu'un seul constructeur qui attend deux paramètres de type String : le nom et la valeur associée.

La classe Cookie possède plusieurs getter et setter pour obtenir ou définir des attributs qui sont tous optionnels.

Attribut	Rôle
Comment	Commentaire associé au cookie
Domain	Nom de domaine (partiel ou complet) associé au cookie. Seuls les serveurs contenant ce nom de domaine recevront le cookie.
MaxAge	Durée de vie en secondes du cookie. Une fois ce délai expiré, le cookie est détruit sur le poste client par le navigateur. Par défaut la valeur limite la durée de vie du cookie à la durée de vie de l'exécution du navigateur
Name	Nom du cookie
Path	Chemin du cookie. Ce chemin permet de renvoyer le cookie uniquement au serveur dont l'url contient également le chemin. Par défaut, cet attribut contient le chemin de l'url de la servlet. Par exemple, pour que le cookie soit renvoyé à toutes les requêtes du serveur, il suffit d'affecter la valeur "/" à cette attribut.
Secure	Booléen qui précise si le cookie ne doit être envoyé que via une connexion SSL.
Value	Valeur associée au cookie.
Version	Version du protocole utilisé pour gérer le cookie

55.6.2. L'enregistrement et la lecture d'un cookie

Pour envoyer un cookie au browser, il suffit d'utiliser la méthode addCookie() de la classe HttpServletResponse.

Exemple :

```
vCookie monCookie = new Cookie("nom","valeur");
response.addCookie(monCookie);
```

Pour lire un cookie envoyé par le browser, il faut utiliser la méthode getCookies() de la classe HttpServletRequest. Cette méthode renvoie un tableau d'objet Cookie. Les cookies sont renvoyés dans l'en-tête de la requête http. Pour rechercher un cookie particulier, il faut parcourir le tableau et rechercher le cookie à partir de son nom grâce à la méthode

getName() de l'objet Cookie.

Exemple :

```
Cookie[] cookies = request.getCookies();
String valeur = "";
for(int i=0;i<cookies.length;i++) {
  if(cookies[i].getName().equals("nom")) {
    valeur=cookies[i].getValue();
  }
}
```

55.7. Le partage d'informations entre plusieurs échanges HTTP

Cette section sera développée dans une version future de ce document

55.8. Packager une application web

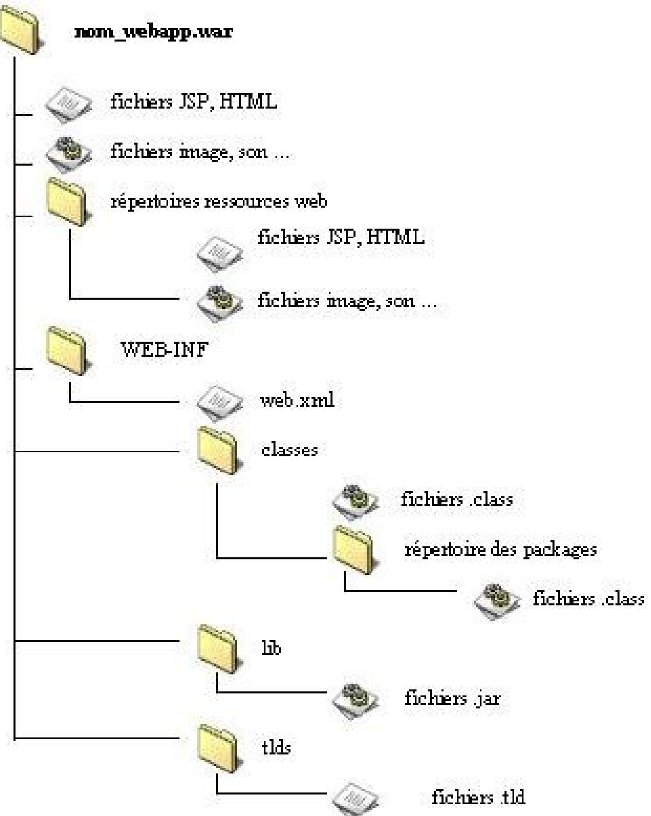
Le format war (Web Application Archive) permet de regrouper en un seul fichier tous les éléments d'une application web que ce soit pour le côté serveur (servlets, JSP, classes java, ...) ou pour le côté client (ressources HTML, images, son ...).

C'est une extension du format jar spécialement dédiée aux applications web qui a été introduite dans les spécifications de la version 2.2 des servlets. C'est un format indépendant de toute plate-forme et exploitable par tous les conteneurs web qui respectent à minima cette version des spécifications.

Le but principal est de simplifier le déploiement d'une application web et d'uniformiser cette action quel que soit le conteneur web utilisé.

55.8.1. La structure d'un fichier .war

Comme les fichiers jar, les fichiers war possèdent une structure particulière qui est incluse dans un fichier compressé de type "zip" possédant comme extension ".war".



Le nom du fichier .war est important car ce nom sera automatiquement associé dans l'url pour l'accès à l'application en concaténant le nom du domaine, un slash et le nom du fichier war. Par exemple, pour un serveur web sur le poste local avec un fichier test.war déployé sur le serveur d'application, l'url pour accéder à l'application web sera http://localhost/test/

Le répertoire WEB-INF et le fichier web.xml qu'il contient doivent obligatoirement être présents dans l'archive. Le fichier web.xml est le descripteur de déploiement de l'application web.

Le serveur web peut avoir accès via le serveur d'application à toutes les ressources contenues dans le fichier .war hormis celles contenues dans le répertoire WEB-INF. Ces dernières ne sont accessibles qu'au serveur d'application.

Le répertoire WEB-INF/classes est automatiquement ajouté par le conteneur au CLASSPATH lors du déploiement de l'application web.

L'archive web peut être créée avec l'outil jar fourni avec le JDK ou avec un outil commercial. Avec l'outil jar, il suffit de créer l'arborescence de l'application, de se placer dans le répertoire racine de cette arborescence et d'exécuter la commande :

jar cvf nom_web_app.war .

Toute l'arborescence avec les fichiers qu'elle contient sera incluse dans le fichier nom_web_app.jar.

55.8.2. Le fichier web.xml

Le fichier /WEB-INF/web.xml est un fichier au format XML qui est le descripteur de déploiement permettant de configurer : l'application, les servlets, les sessions, les bibliothèques de tags personnalisées, les paramètres de contexte, les types Mimes, les pages par défaut, les ressources externes, la sécurité de l'application et des ressources J2EE.

Le fichier web.xml commence par un prologue et une indication sur la version de la DTD à utiliser. Celle-ci dépend des spécifications de l'API servlet utilisée.

Exemple : servlet 2.2

```
<?xml version="1.0" encoding="ISO-8859-1"?>
  <!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
  "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
```

Exemple : servlet 2.3

```
  <?xml version="1.0" encoding="ISO-8859-1"?>
  <!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">
```

L'élément racine est le tag <web-app>. Cet élément peut avoir plusieurs tags fils dont l'ordre d'utilisation doit respecter celui défini dans la DTD utilisée.

Le tag <icon> permet de préciser une petite et une grande image qui pourront être utilisées par des outils graphiques.

Le tag <display-name> permet de donner un nom pour l'affichage dans les outils.

Le tag <description> permet de fournir un texte de description de l'application web.

Le tag <context-param> permet de fournir un paramètre d'initialisation de l'application. Ce tag peut avoir trois tags fils : <param-name>, <param-value> et <description>. Il doit y en avoir autant que de paramètres d'initialisation. Les valeurs fournies peuvent être retrouvées dans le code de la servlet grâce à la méthode getInitParameter() de l'objet ServletContext.

Le tag <servlet> permet de définir une servlet. Le tag fils <icon> permet de préciser une petite et une grande image pour les outils graphique. Le tag <servlet-name> permet de donner un nom à la servlet qui sera utilisé pour le mapping avec l'URL par défaut de la servlet. Le tag <display-name> permet de donner un nom d'affichage. Le tag <description> permet de fournir une description de la servlet. Le tag <servlet-class> permet de préciser le nom complètement qualifié de la classe java dont la servlet sera une instance. Le tag <init-param> permet de préciser un paramètre d'initialisation pour la servlet. Ce tag possède les tags fils <param-name>, <param-value>, <description>. Les valeurs fournies peuvent être retrouvées dans le code de la servlet grâce à la méthode getInitParameter() de la classe ServletConfig. Le tag <load-on-startup> permet de préciser si la servlet doit être instanciée lors de l'initialisation du conteneur. Il est possible de préciser dans le corps de ce tag un numéro de séquence qui permettra d'ordonner la création des servlets.

Exemple : servlet 2.2

```
  <servlet>
    <servlet-name>MaServlet</servlet-name>
    <servlet-class>com.jmdoudoux.test.servlet.MaServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
    <init-param>
                  <param-name>param1</param-name>
                        <param-value>valeur1</param-value>
    </init-param>

  </servlet>
```

Le tag <servlet-mapping> permet d'associer la servlet à une URL. Ce tag possède les tags fils <servlet-name> et <servlet-mapping>.

Exemple : servlet 2.2

```
  <servlet-mapping>
    <servlet-name>MaServlet</servlet-name>
          <url-pattern>/test</url-pattern>
  </servlet-mapping>
```

Le tag <session-config> permet de configurer les sessions. Le tag fils <session-timeout> permet de préciser la durée maximum d'inactivité de la session avant sa destruction. La valeur fournie dans le corps de ce tag est exprimé en minutes.

Exemple : servlet 2.2

```
<session-config>
  <session-timeout>15</session-timeout>
      </session-config>
```

Le tag <mime-mapping> permet d'associer des extensions à un type mime particulier.

Le tag <welcome-file-list> permet de définir les pages par défaut. Chacun des fichiers est défini grâce au tag fils <welcome-file>

Exemple : servlet 2.2

```
<welcome-file-list>
  <welcome-file>index.jsp</welcome-file>
  <welcome-file>index.htm</welcome-file>
<welcome-file-list>
```

Le tag <error-page> permet d'associer une page web à un code d'erreur HTTP particulier ou a une exception java particulière. Le code erreur est précisé avec le tag fils <error-code>. L'exception Java est précisée avec le tag fils <exception-type>. La page web est précisée avec le tag fils <location>.

Le tag <tag-lib> permet de définir une bibliothèque de tags personnalisée. Le tag fils <taglib-uri> permet de préciser l'URI de la bibliothèque. Le tag fils <taglib-location> permet de préciser le chemin de la bibliothèque.

Exemple : déclaration de la biliothèque core de JSTL

```
<taglib>
        <taglib-uri>http://java.sun.com/jstl/core</taglib-uri>
        <taglib-location>/WEB-INF/tld/c.tld</taglib-location>
</taglib>
```

55.8.3. Le déploiement d'une application web

Le déploiement d'une archive web dans un serveur d'application est très facile car il suffit simplement de copier le fichier .war dans le répertoire par défaut dédié aux applications web. Par exemple dans Tomcat, c'est le répertoire webapps. Attention cependant, si chaque conteneur qui respecte les spécifications 1.1 des JSP sait utiliser un fichier .war, leur exploitation par chaque conteneur est légèrement différente.

Par exemple avec Tomcat, il est possible de travailler directement dans le répertoire webapps avec le contenu de l'archive web décompressé. Cette fonctionnalité est particulièrement intéressante lors de la phase de développement de l'application car il n'est alors pas obligatoire de générer l'archive web à chaque modification pour réaliser des tests. Attention, si l'application est redéployée sous la forme d'une archive .war, il faut obligatoirement supprimer le répertoire qui contient l'ancienne version de l'application.

55.9. L'utilisation Log4J dans une servlet

Log4J est un framework dont le but est de faciliter la mise en oeuvre de fonctionnalités de logging dans une application. Il est notamment possible de l'utiliser dans une application web. Pour plus de détails sur cette API, consultez la section qui lui est consacrée dans le chapitre «Logging» de ce didacticiel.

Pour utiliser Log4J dans une application web, il est nécessaire d'initialiser Log4J avant utilisation. Le plus simple est d'écrire une servlet qui va réaliser cette initialisation et qui sera chargée automatiquement au chargement de l'application web.

Dans la méthode init() de la servlet, deux paramètres sont récupérés et sont utilisés pour

- définir une variable d'environnement qui sera utilisée par Log4J dans son fichier de configuration pour définir le chemin du fichier journal utilisé
- initialiser Log4J en utilisant un fichier de configuration

Exemple :

```
package com.jmd.test.log4j;

import org.apache.log4j.PropertyConfigurator;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.*;

public class InitServlet extends HttpServlet {
        public void init() {

                String cheminWebApp = getServletContext().getRealPath("/");
                String cheminLogConfig = cheminWebApp + getInitParameter("log4j-fichier-config");
                String cheminLog = cheminWebApp + getInitParameter("log4j-chemin-log");

                File logPathDir = new File( cheminLog );
                System.setProperty( "log.chemin", cheminLog );

                if (cheminLogConfig != null) {
                        PropertyConfigurator.configure(cheminLogConfig);
                }
        }

        public void doGet(HttpServletRequest req, HttpServletResponse res) {
        }
}
```

Dans le fichier web.xml, il faut configurer les servlets utilisées et notamment la servlet définie pour initialiser Log4J. Celle ci attend au moins deux paramètres :

- log4j-fichier-config : ce paramètre doit avoir comme valeur le chemin relatif du fichier de configuration de Log4J par rapport à la racine de l'application web
- log4j-chemin-log : ce paramètre doit avoir comme valeur le chemin du répertoire qui va contenir les fichiers journaux générés par Log4J par rapport à la racine de l'application web

Exemple : le fichier web.xml

```
<?xml version="1.0"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <servlet>
    <servlet-name>initservlet</servlet-name>
    <servlet-class>com.jmd.test.log4j.InitServlet</servlet-class>
    <init-param>
      <param-name>log4j-fichier-config</param-name>
      <param-value>WEB-INF/classes/log4j.properties</param-value>
    </init-param>
    <init-param>
      <param-name>log4j-chemin-log</param-name>
      <param-value>WEB-INF/log</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet>
    <servlet-name>TestServlet</servlet-name>
    <servlet-class>com.jmd.test.log4j.TestServlet</servlet-class>
  </servlet>

  <servlet-mapping>
```

```
    <servlet-name>TestServlet</servlet-name>
    <url-pattern>/test</url-pattern>
  </servlet-mapping>
</web-app>
```

Il est important de demander le chargement automatique de la servlet en donnant la valeur 1 au tag <load-on-startup> de la servlet.

Il faut définir le fichier de configuration nommé par exemple log4j.properties et le placer dans le répertoire WEB-INF/classes de l'application.

Exemple : le fichier log4j.properties

```
# initialisation de la racine du logger avec le niveau INFO
log4j.rootLogger=INFO, A1

# utilisation d'un fichier pour stocker les informations du journal
log4j.appender.A1=org.apache.log4j.FileAppender
log4j.appender.A1.file=${log.chemin}/application.log

# utilisation du layout de base
log4j.appender.A1.layout=org.apache.log4j.SimpleLayout
```

L'utilisation de Log4J dans une servlet est alors équivalente à celle d'une application standalone.

Exemple : une servlet qui utilise Log4J

```
package com.jmd.test.log4j;

import java.io.IOException;
import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.log4j.Logger;

public class TestServlet extends HttpServlet {

  private static final Logger logger = Logger.getLogger(TestServlet.class);

  public void init(ServletConfig config) throws ServletException {
    super.init(config);
    logger.info("initialisation de la servet TestServlet");
  }

  public void doGet(HttpServletRequest req, HttpServletResponse res) {
    StringBuffer sb = new StringBuffer();

    logger.debug("appel doGet de la servlet TestServlet");

    sb.append("<HTML>\n");
    sb.append("<HEAD>\n");
    sb.append("<TITLE>Bonjour</TITLE>\n");
    sb.append("</HEAD>\n");
    sb.append("<BODY>\n");
    sb.append("<H1>Bonjour</H1>\n");
    sb.append("</BODY>\n");
    sb.append("</HTML>");

    res.setContentType("text/html");
    res.setContentLength(sb.length());

    try {
      res.getOutputStream().print(sb.toString());
    } catch (IOException e) {
      e.printStackTrace();
    }
```

```
    }
}
```

Lors de l'exécution de l'application web, le journal est rempli dans le fichier /WEB-INF/log/application.log.

56. Les JSP (Java Server Pages)

Les JSP (Java Server Pages) sont une technologie Java qui permettent la génération de pages web dynamiques.

La technologie JSP permet de séparer la présentation sous forme de code HTML et les traitements sous formes de classes Java définissant un bean ou une servlet. Ceci est d'autant plus facile que les JSP définissent une syntaxe particulière permettant d'appeler un bean et d'insérer le résultat de son traitement dans la page HTML dynamiquement.

Les informations fournies dans ce chapitre concernent les spécifications 1.0 et ultérieures des JSP.

Ce chapitre contient plusieurs sections :

- La présentation des JSP
- Les outils nécessaires
- Le code HTML
- Les Tags JSP
- Un Exemple très simple
- La gestion des erreurs
- Les bibliothèques de tags personnalisés (custom taglibs)

56.1. La présentation des JSP

Les JSP permettent d'introduire du code Java dans des tags prédéfinis à l'intérieur d'une page HTML. La technologie JSP mélange la puissance de Java côté serveur et la facilité de mise en page d'HTML côté client.

Sun fourni de nombreuses informations sur la technologie JSP à l'adresse suivante : http://java.sun.com/products/jsp/index.html

Une JSP est habituellement constituée :

- de données et de tags HTML
- de tags JSP
- de scriptlets (code Java intégré à la JSP)

Les fichiers JSP possèdent par convention l'extension .jsp.

Concrètement, les JSP sont basées sur les servlets. Au premier appel de la page JSP, le moteur de JSP génère et compile automatiquement une servlet qui permet la génération de la page web. Le code HTML est repris intégralement dans la servlet. Le code Java est inséré dans la servlet.

La servlet générée est compilée et sauvegardée puis elle est exécutée. Les appels suivants de la JSP sont beaucoup plus rapides car la servlet, conservée par le serveur, est directement exécutée.

Il y a plusieurs manières de combiner les technologies JSP, les beans/EJB et les servlets en fonction des besoins pour développer des applications web.

Comme le code de la servlet est généré dynamiquement, les JSP sont relativement difficiles à déboguer.

Cette approche possède plusieurs avantages :

- l'utilisation de Java par les JSP permet une indépendance de la plate-forme d'exécution mais aussi du serveur web utilisé.
- la séparation des traitements et de la présentation : la page web peut être écrite par un designer et les tags Java peuvent être ajoutés ensuite par le développeur. Les traitements peuvent être réalisés par des composants réutilisables (des Java beans).
- les JSP sont basées sur les servlets : tout ce qui est fait par une servlet pour la génération de pages dynamiques peut être fait avec une JSP.

Il existe plusieurs versions des spécifications JSP :

Version	
0.91	Première release
1.0	Juin 1999 : première version finale
1.1	Décembre 1999 :
1.2	Octobre 2000, JSR 053
2.0	JSR 152

56.1.1. Le choix entre JSP et Servlets

Les servlets et les JSP ont de nombreux points communs puisque qu'une JSP est finalement convertie en une servlet. Le choix d'utiliser l'une ou l'autre de ces technologies ou les deux doit être fait pour tirer le meilleur parti de leurs avantages.

Dans une servlet, les traitements et la présentation sont regroupés. L'aspect présentation est dans ce cas pénible à développer et à maintenir à cause de l'utilisation répétitive de méthodes pour insérer le code HTML dans le flux de sortie. De plus, une simple petite modification dans le code HTML nécessite la recompilation de la servlet. Avec un JSP, la séparation des traitements et de la présentation rend ceci très facile et automatique.

Il est préférable d'utiliser les JSP pour générer des pages web dynamiques.

L'usage des servlets est obligatoire si celles ci doivent communiquer directement avec une applet ou une application et non plus avec un serveur web.

56.1.2. Les JSP et les technologies concurrentes

Il existe plusieurs technologies dont le but est similaire aux JSP notamment ASP, PHP et ASP.Net. Chacune de ces technologies possèdent des avantages et des inconvénients dont voici une liste non exhaustive.

	JSP	PHP	ASP	ASP.Net
langage	Java	PHP	VBScript ou JScript	Tous les langages supportés par .Net (C#, VB.Net, Delphi, ...)
mode d'exécution	Compilé en pseudo code (byte code)	Interprété	Interprété	Compilé en pseudo code (MSIL)
principaux avantages	Repose sur la plate-forme Java dont elle hérite des avantages	Open source Nombreuses bibliothèques et sources d'applications libres disponibles Facile à mettre en	Facile à mettre en oeuvre	Repose sur la plate-forme .Net dont elle hérite des avantages Wysiwyg et événementiel

		oeuvre		Code behind pour séparation affichage / traitements
principaux inconvénients	Débogage assez fastidieux Beaucoup de code à écrire	Débogage assez fastidieux Beaucoup de code à écrire support partiel de la POO en attendant la version 5	Débogage assez fastidieux Beaucoup de code à écrire Fonctionne essentiellement sur plateformes Windows Pas de POO, objet métier encapsulé dans des objets COM lourd à mettre en oeuvre	Fonctionne essentiellement sur plate-formes Windows. (Voir le projet Mono pour le support d'autres plateformes)

56.2. Les outils nécessaires

Dans un premier temps, Sun a fourni un kit de développement pour les JSP : le Java Server Web Development Kit (JSWDK). Actuellement, Sun a chargé le projet Apache de développer l'implémentation officielle d'un moteur de JSP. Ce projet se nomme Tomcat.

En fonction des versions des API utilisées, il faut choisir un produit différent. Le tableau ci dessous résume le produit à utiliser en fonction de la version des API mise en oeuvre.

Produit	Version	Version de l'API servlet implémentée	Version de l'API JSP implémentée
JSWDK	1.0.1	2.1	1.0
Tomcat	3.2	2.2	1.1
Tomcat	4.0	2.3	1.2
Tomcat	5.0	2.4	2.0

Ces produits sont librement téléchargeables sur le site de Sun à l'adresse suivante :
http://java.sun.com/products/jsp/download.html

Pour télécharger le JSWDK, il faut cliquer sur le lien " archive ".

Il est aussi possible d'utiliser n'importe quel conteneur web compatible avec les spécifications de la plate-forme J2EE. Une liste non exhaustive est fournie dans le chapitre «Les outils libres et commerciaux».

56.2.1. L'outil JavaServer Web Development Kit (JSWDK) sous Windows

Le JSWDK est proposé sous la forme d'un fichier zip nommé jswdk_1_0_1-win.zip.

Pour l'installer, il suffit de décompresser l'archive dans un répertoire du système. Pour lancer le serveur, il suffit d'exécuter le fichier startserver.bat

Pour lancer le serveur :

```
C:\jswdk-1.0.1>startserver.bat

Using classpath:.\classes;.\webserver.jar;.\lib\jakarta.jar;.\lib\servlet.jar;.
\lib\jsp.jar;.\lib\jspengine.jar;.\examples\WEB-INF\jsp\beans;.\webpages\WEB-INF
\servlets;.\webpages\WEB-INF\jsp\beans;.\lib\xml.jar;.\lib\moo.jar;\lib\tools.ja
```

```
r;C:\jdk1.3\lib\tools.jar;
C:\jswdk-1.0.1>
```

Le serveur s'exécute dans une console en tache de fond. Cette console permet de voir les messages émis par le serveur.

Exemple : au démarrage

```
JSWDK WebServer Version 1.0.1
Loaded configuration from: file:C:\jswdk-1.0.1\webserver.xml
endpoint created: localhost/127.0.0.1:8080
```

Si la JSP contient une erreur, le serveur envoie une page d'erreur :

Une exception est levée et est affichée dans la fenêtre ou le serveur s'exécute :

Exemple :

```
-- Commentaires de la page JSP --
^
1 error
at com.sun.jsp.compiler.Main.compile(Main.java:347)
at com.sun.jsp.runtime.JspLoader.loadJSP(JspLoader.java:135)
at com.sun.jsp.runtime.JspServlet$JspServletWrapper.loadIfNecessary(JspS
ervlet.java:77)
at com.sun.jsp.runtime.JspServlet$JspServletWrapper.service(JspServlet.j
ava:87)
at com.sun.jsp.runtime.JspServlet.serviceJspFile(JspServlet.java:218)
at com.sun.jsp.runtime.JspServlet.service(JspServlet.java:294)
at javax.servlet.http.HttpServlet.service(HttpServlet.java:840)
at com.sun.web.core.ServletWrapper.handleRequest(ServletWrapper.java:155
)
at com.sun.web.core.Context.handleRequest(Context.java:414)
at com.sun.web.server.ConnectionHandler.run(ConnectionHandler.java:139)
HANDLER THREAD PROBLEM: java.io.IOException: Socket Closed
java.io.IOException: Socket Closed
at java.net.PlainSocketImpl.getInputStream(Unknown Source)
at java.net.Socket$1.run(Unknown Source)
at java.security.AccessController.doPrivileged(Native Method)
at java.net.Socket.getInputStream(Unknown Source)
at com.sun.web.server.ConnectionHandler.run(ConnectionHandler.java:161)
```

Le répertoire work contient le code et le byte code des servlets générées à partir des JSP.

Pour arrêter le serveur, il suffit d'exécuter le script stopserver.bat.

A l'arrêt du serveur, le répertoire work qui contient les servlets générées à partir des JSP est supprimé.

56.2.2. Le serveur Tomcat

La mise en oeuvre et l'utilisation de Tomcat est détaillée dans une section du chapitre «Les servlets».

56.3. Le code HTML

Une grande partie du contenu d'une JSP est constituée de code HTML. D'ailleurs, le plus simple pour écrire une JSP est d'écrire le fichier HTML avec un outil dédié et d'ajouter ensuite les tags JSP pour ce qui concerne les parties dynamiques.

La seule restriction concernant le code HTML concerne l'utilisation dans la page générée du texte " <% " et " %> ". Dans ce cas, le plus simple est d'utiliser les caractères spéciaux HTML < et >. Sinon l'analyseur syntaxique du moteur de JSP considère que c'est un tag JSP et renvoie une erreur.

Exemple :

```
<HTML>
<HEAD>
<TITLE>Test</TITLE>
</HEAD>
<BODY>
<p>Plusieurs tags JSP commencent par &lt;% et se finissent par %&gt;</p>
</BODY>
</HTML>
```

56.4. Les Tags JSP

Il existe trois types de tags :

- tags de directives : ils permettent de contrôler la structure de la servlet générée
- tags de scripting: ils permettent d'insérer du code Java dans la servlet
- tags d'actions: ils facilitent l'utilisation de composants

Attention : Les noms des tags sont sensibles à la casse.

56.4.1. Les tags de directives <%@ ... %>

Les directives permettent de préciser des informations globales sur la page JSP. Les spécifications des JSP définissent trois directives :

- page : permet de définir des options de configuration
- include : permet d'inclure des fichiers statiques dans la JSP avant la génération de la servlet
- taglib : permet de définir des tags personnalisés

Leur syntaxe est la suivante :

<%@ *directive attribut*="*valeur*" ... %>

56.4.1.1. La directive page

Cette directive doit être utilisée dans toutes les pages JSP : elle permet de définir des options qui s'appliquent à toute la JSP.

Elle peut être placée n'importe où dans le source mais il est préférable de la mettre en début de fichier, avant même le tag <HTML>. Elle peut être utilisée plusieurs fois dans une même page mais elle ne doit définir la valeur d'une option qu'une seule fois, sauf pour l'option import.

Les options définies par cette directive sont de la forme option=valeur.

Option	Valeur	Valeur par défaut	Autre valeur possible
autoFlush	Une chaîne	«true»	«false»
buffer	Une chaîne	«8kb»	«none» ou «nnnkb» (nnn indiquant la valeur)
contentType	Une chaîne contenant le type mime		
errorPage	Une chaîne contenant une URL		
extends	Une classe		
import	Une classe ou un package.*		
info	Une chaîne		
isErrorPage	Une chaîne	«false»	«true»
isThreadSafe	Une chaîne	«true»	«false»
langage	Une chaîne	«java»	
session	Une chaîne	«true»	«false»

Exemple :

```
<%@ page import="java.util.*" %>
<%@ page import="java.util.Vector" %>
<%@ page info="Ma premiere JSP"%>
```

Les options sont :

- autoFlush="**true**|false"

  Cette option indique si le flux en sortie de la servlet doit être vidé quand le tampon est plein. Si la valeur est false, une exception est levée dès que le tampon est plein. On ne peut pas mettre false si la valeur de buffer est none.

- buffer="none|**8kb**|*size*kb"

  Cette option permet de préciser la taille du buffer des données générées contenues par l'objet out de type JspWriter.

- contentType="*mimeType* [; charset=*characterSet*]" | "**text/html;charset=ISO-8859-1**"

  Cette option permet de préciser le type MIME des données générées.

  Cette option est équivalente à <% response.setContentType("mimeType"); %>

- errorPage=*"relativeURL"*

  Cette option permet de préciser la JSP appelée au cas où une exception est levée

  Si l'URL commence pas un '/', alors l'URL est relative au répertoire principale du serveur web sinon elle est relative au répertoire qui contient la JSP

- extends=*"package.class"*

  Cette option permet de préciser la classe qui sera la super classe de l'objet Java créé à partir de la JSP.

- import= "{ *package.class* | *package.** }, ..."

  Cette option permet d'importer des classes contenues dans des packages utilisées dans le code de la JSP. Cette option s'utilise comme l'instruction import dans un code source Java.

  Chaque classe ou package est séparée par une virgule.

  Cette option peut être présente dans plusieurs directives page.

- info="text"

  Cette option permet de préciser un petit descriptif de la JSP. Le texte fourni sera renvoyé par la méthode getServletInfo() de la servlet générée.

- isErrorPage="true|**false**"

  Cette option permet de préciser si la JSP génère une page d'erreur. La valeur true permet d'utiliser l'objet Exception dans la JSP

- isThreadSafe="**true**|false"

  Cette option indique si la servlet générée sera multithread : dans ce cas, une même instance de la servlet peut gérer plusieurs requêtes simultanément. En contre partie, elle doit gérer correctement les accès concurrents aux ressources. La valeur false impose à la servlet générée d'implémenter l'interface SingleThreadModel.

- language="java"

  Cette option définit le langage utilisé pour écrire le code dans la JSP. La seule valeur autorisée actuellement est «java».

- session="**true**|false"

  Cette option permet de préciser si la JSP est incluse dans une session ou non. La valeur par défaut (true) permet l'utilisation d'un objet session de type HttpSession qui permet de gérer des informations dans une session.

56.4.1.2. La directive include

Cette directive permet d'inclure un fichier dans le code source JSP. Le fichier inclus peut être un fragment de code JSP, HTML ou Java. Le fichier est inclus dans la JSP avant que celle ci ne soit interprétée par le moteur de JSP.

Ce tag est particulièrement utile pour insérer un élément commun à plusieurs pages tel qu'un en-tête ou un bas de page.

Si le fichier inclus est un fichier HTML, celui ci ne doit pas contenir de tag <HTML>, </HTML>, <BODY> ou </BODY> qui ferait double emploi avec ceux présents dans le fichier JSP. Ceci impose d'écrire des fichiers HTML particuliers uniquement pour être inclus dans les JSP : ils ne pourront pas être utilisés seuls.

La syntaxe est la suivante :

<%@ include file="chemin relatif du fichier" %>

Si le chemin commence par un '/', alors le chemin est relatif au contexte de l'application, sinon il est relatif au fichier JSP.

Exemple :

```
bonjour.htm :

<p><table border="1" cellpadding="4" cellspacing="0" width="30%" align=center >
<tr bgcolor="#A6A5C2">
<td align="center">BONJOUR</Td>
</Tr>
```

```
</table></p>
```

Exemple : Test1.jsp

```
<HTML>
<HEAD>
<TITLE>Essai de page JSP</TITLE>
</HEAD>
<BODY>
<p align="center">Test d'inclusion d'un fichier dans la JSP</p>
<%@ include file="bonjour.htm"%>
<p align="center">fin</p>
</BODY>
</HTML>
```

Pour tester cette JSP avec le JSWDK, il suffit de placer ces deux fichiers dans le répertoire jswdk-1.0.1\examples\jsp\test.

Pour visualiser la JSP, il faut saisir l'url http://localhost:8080/examples/jsp/test/Test1.jsp dans un navigateur.

Attention : un changement dans le fichier inclus ne provoque pas une regénération et une compilation de la servlet correspondant à la JSP. Pour insérer un fichier dynamiquement à l'exécution de la servlet il faut utiliser le tag <jsp:include>.

56.4.1.3. La directive taglib

Cette directive permet de déclarer l'utilisation d'une bibliothèque de tags personnalisés. L'utilisation de cette directive est détaillée dans la section consacrée aux bibliothèques de tags personnalisés.

56.4.2. Les tags de scripting

Ces tags permettent d'insérer du code Java qui sera inclus dans la servlet générée à partir de la JSP. Il existe trois tags pour insérer du code Java :

- le tag de déclaration : le code Java est inclus dans le corps de la servlet générée. Ce code peut être la déclaration de variables d'instances ou de classes ou la déclaration de méthodes.
- le tag d'expression : évalue une expression et insère le résultat sous forme de chaîne de caractères dans la page web générée.
- le tag de scriptlets : par défaut, le code Java est inclus dans la méthode service() de la servlet.

Il est possible d'utiliser dans ces tags plusieurs objets définis par les JSP.

56.4.2.1. Le tag de déclarations <%! ... %>

Ce tag permet de déclarer des variables ou des méthodes qui pourront être utilisées dans la JSP. Il ne génère aucun caractère dans le fichier HTML de sortie.

La syntaxe est la suivante :

<%! declarations %>

Exemple :

```
<%! int i = 0; %>
<%! dateDuJour = new java.util.Date(); %>
```

Les variables ainsi déclarées peuvent être utilisées dans les tags d'expressions et de scriptlets.

Il est possible de déclarer plusieurs variables dans le même tag en les séparant avec des caractères ' ;'.

Ce tag permet aussi d'insérer des méthodes dans le corps de la servlet.

Exemple :

```
<HTML>
<HEAD>
<TITLE>Test</TITLE>
</HEAD>
<BODY>
<%!
int minimum(int val1, int val2) {
   if (val1 < val2) return val1;
   else return val2;
}
%>
<% int petit = minimum(5,3);%>
<p>Le plus petit de 5 et 3 est <%= petit %></p>
</BODY>
</HTML>
```

56.4.2.2. Le tag d'expressions <%= ... %>

Le moteur de JSP remplace ce tag par le résultat de l'évaluation de l'expression présente dans le tag.

Ce résultat est toujours converti en une chaîne. Ce tag est un raccourci pour éviter de faire appel à la méthode println() lors de l'insertion de données dynamiques dans le fichier HTML.

La syntaxe est la suivante :

<%= expression %>

Le signe '=' doit être collé au signe '%'.

Attention : il ne faut pas mettre de ' ;' à la fin de l'expression.

Exemple : Insertion de la date dans la page HTML

```
<%@ page import="java.util.*" %>
<HTML>
<HEAD>
<TITLE>Essai de page JSP</TITLE>
</HEAD>
<BODY>
<p align="center">Date du jour :
<%= new Date() %>
</p>
</BODY>
</HTML>
```

Résultat :

```
Date du jour : Thu Feb 15 11:15:24 CET 2001
```

L'expression est évaluée et convertie en chaîne avec un appel à la méthode toString(). Cette chaîne est insérée dans la page HTML en remplacement du tag. Il est ainsi possible que le résultat soit une partie ou la totalité d'un tag HTML ou même une JSP.

Exemple :

```
<HTML>
<HEAD>
<TITLE>Essai de page JSP</TITLE>
</HEAD>
<BODY>
<%="<H1>" %>
Bonjour
<%="</H1>" %>
</BODY>
</HTML>
```

Résultat : code HTML généré

```
<HTML>
<HEAD>
<TITLE>Essai de page JSP</TITLE>
</HEAD>
<BODY>
<H1>
Bonjour
</H1>
</BODY>
</HTML>
```

56.4.2.3. Les variables implicites

Les spécifications des JSP définissent plusieurs objets utilisables dans le code dont les plus utiles sont :

Object	Classe	Rôle
out	javax.servlet.jsp.JspWriter	Flux en sortie de la page HTML générée
request	javax.servlet.http.HttpServletRequest	Contient les informations de la requête
response	javax.servlet.http.HttpServletResponse	Contient les informations de la réponse
session	javax.servlet.http.HttpSession	Gère la session

56.4.2.4. Le tag des scriptlets <% ... %>

Ce tag contient du code Java nommé un scriptlet.

La syntaxe est la suivante : <% code Java %>

Exemple :

```
<%@ page import="java.util.Date"%>
<html>
<body>
<%! Date dateDuJour; %>
<% dateDuJour = new Date();%>
Date du jour : <%= dateDuJour %><BR>
</body>
</html>
```

Par défaut, le code inclus dans le tag est inséré dans la méthode service() de la servlet générée à partir de la JSP.

Ce tag ne peut pas contenir autre chose que du code Java : il ne peut pas par exemple contenir de tags HTML ou JSP. Pour faire cela, il faut fermer le tag du scriptlet, mettre le tag HTML ou JSP puis de nouveau commencer un tag de scriptlet pour continuer le code.

Exemple :

```
<HTML>
<HEAD>
<TITLE>Essai de page JSP</TITLE>
</HEAD>
<BODY>
<% for (int i=0; i<10; i++) { %>
<%= i %> <br>
<% }%>
</BODY>
</HTML>
```

Résultat : la page HTML générée

```
<HTML>
<HEAD>
<TITLE>Essai de page JSP</TITLE>
</HEAD>
<BODY>
0 <br>
1 <br>
2 <br>
3 <br>
4 <br>
5 <br>
6 <br>
7 <br>
8 <br>
9 <br>
</BODY>
</HTML>
```

56.4.3. Les tags de commentaires

Il existe deux types de commentaires avec les JSP :

- les commentaires visibles dans le code HTML
- les commentaires invisibles dans le code HTML

56.4.3.1. Les commentaires HTML <!-- ... -->

Ces commentaires sont ceux définis par format HTML. Ils sont intégralement reconduits dans le fichier HTML généré. Il est possible d'insérer, dans ce tag, un tag JSP de type expression qui sera exécuté.

La syntaxe est la suivante :

<!-- commentaires [<%= expression %>] -->

Exemple :

```
<%@ page import="java.util.*" %>
<HTML>
<HEAD>
<TITLE>Essai de page JSP</TITLE>
</HEAD>
<BODY>
<!-- Cette page a ete generee le <%= new Date() %> -->
<p>Bonjour</p>
</BODY>
</HTML>
```

Résultat :

```
<HTML>
<HEAD>
<TITLE>Essai de page JSP</TITLE>
</HEAD>
<BODY>
<!-- Cette page a ete generee le Thu Feb 15 11:44:25 CET 2001 -->
<p>Bonjour</p>
</BODY>
</HTML>
```

Le contenu d'une expression incluse dans des commentaires est dynamique : sa valeur peut changer à chaque génération de la page en fonction de son contenu.

56.4.3.2. Les commentaires cachés <%-- ... --%>

Les commentaires cachés sont utilisés pour documenter la page JSP. Leur contenu est ignoré par le moteur de JSP et ne sont donc pas reconduits dans la page HTML générée.

La syntaxe est la suivante :

<%-- commentaires --%>

Exemple :

```
<HTML>
<HEAD>
<TITLE>Essai de page JSP</TITLE>
</HEAD>
<BODY>
<%-- Commentaires de la page JSP --%>
<p>Bonjour</p>
</BODY>
</HTML>
```

Résultat :

```
<HTML>
<HEAD>
<TITLE>Essai de page JSP</TITLE>
</HEAD>
<BODY>
<p>Bonjour</p>
</BODY>
</HTML>
```

Ce tag peut être utile pour éviter l'exécution de code lors de la phase de débogage.

56.4.4. Les tags d'actions

Les tags d'actions permettent de réaliser des traitements couramment utilisés.

56.4.4.1. Le tag <jsp:useBean>

Le tag <jsp:useBean> permet de localiser une instance ou d'instancier un bean pour l'utiliser dans la JSP.

L'utilisation d'un bean dans une JSP est très pratique car il peut encapsuler des traitements complexes et être réutilisable par d'autre JSP ou composants. Le bean peut par exemple assurer l'accès à une base de données. L'utilisation des beans

permet de simplifier les traitements inclus dans la JSP.

Lors de l'instanciation d'un bean, on précise la porté du bean. Si le bean demandé est déjà instancié pour la portée précisée alors il n'y pas de nouvelle instance du bean qui est créée mais sa référence est simplement renvoyée : le tag <jsp:useBean> n'instancie donc pas obligatoirement un objet.

Ce tag ne permet pas de traiter directement des EJB.

La syntaxe est la suivante :

<jsp:useBean
id="*beanInstanceNam*e"
scope="**page**|request|session|application"
{ class="*package.class*" |
type="*package.class*" |
class="*package.class*" type="*package.class*" |
beanName="{*package.class* | <%= *expression* %>}" type="*package.class*"
}
{ /> |
> ...
</jsp:useBean>
}

L'attribut id permet de donner un nom à la variable qui va contenir la référence sur le bean.

L'attribut scope permet de définir la portée durant laquelle le bean est défini et utilisable. La valeur de cet attribut détermine la manière dont le tag localise ou instancie le bean. Les valeurs possibles sont :

Valeur	Rôle
page	Le bean est utilisable dans toute la page JSP ainsi que dans les fichiers statiques inclus. C'est la valeur par défaut.
request	le bean est accessible durant la durée de vie de la requête. La méthode getAttribute() de l'objet request permet d'obtenir une référence sur le bean.
session	le bean est utilisable par toutes les JSP qui appartiennent à la même session que la JSP qui a instanciée le bean. Le bean est utilisable tout au long de la session par toutes les pages qui y participent. La JSP qui créé le bean doit avoir l'attribut session = « true » dans sa directive page.
application	le bean est utilisable par toutes les JSP qui appartiennent à la même application que la JSP qui a instanciée le bean. Le bean n'est instancié que lors du rechargement de l'application.

L'attribut class permet d'indiquer la classe du bean.

L'attribut type permet de préciser le type de la variable qui va contenir la référence du bean. La valeur indiquée doit obligatoirement être une super classe du bean ou une interface implémentée par le bean (directement ou par héritage)

L'attribut beanName permet d'instancier le bean grâce à la méthode instanciate() de la classe Beans.

Exemple :

```
<jsp:useBean id="monBean" scope="session" class="test.MonBean" />
```

Dans cet exemple, une instance de MonBean est créée une seule et unique fois lors de la session. Dans la même session, l'appel du tag <jsp:useBean> avec le même bean et la même portée ne feront que renvoyer l'instance créée. Le bean est ainsi accessible durant toute la session.

Le tag <jsp:useBean> recherche si une instance du bean existe avec le nom et la portée précisée. Si elle n'existe pas, alors une instance est créée. Si il y a instanciation du bean, alors les tags <jsp:setProperty> inclus dans le tag sont utilisés pour initialiser les propriétés du bean sinon ils sont ignorés. Les tags inclus entre les tags <jsp:useBean> et </jsp:useBean> ne

sont exécutés que si le bean est instancié.

Exemple :

```
<jsp:useBean id="monBean" scope="session" class="test.MonBean" >
<jsp:setProperty name="monBean" property="*" />
</jsp:useBean>
```

Cet exemple a le même effet que le précédent avec une initialisation des propriétés du bean lors de son instanciation avec les valeurs des paramètres correspondants.

Exemple complet : TestBean.jsp

```
<html>
<HEAD>
<TITLE>Essai d'instanciation d'un bean dans une JSP</TITLE>
</HEAD>
<body>
<p>Test d'utilisation d'un Bean dans une JSP </p>
<jsp:useBean id="personne" scope="request" class="test.Personne" />
<p>nom initial = <%=personne.getNom() %></p>
<%
personne.setNom("mon nom");
%>
<p>nom mise à jour = <%= personne.getNom() %></p>
</body>
</html>
```

Exemple complet : Personne.java

```
package test;
public class Personne {
  private String nom;
  private String prenom;

  public Personne() {
    this.nom = "nom par defaut";
    this.prenom = "prenom par defaut";
  }

  public void setNom (String nom) {
    this.nom = nom;
  }

  public String getNom() {
    return (this.nom);
  }

  public void setPrenom (String prenom) {
    this.prenom = prenom;
  }

  public String getPrenom () {
    return (this.prenom);
  }
}
```

Selon le moteur de JSP utilisé, les fichiers du bean doivent être placés dans un répertoire particulier pour être accessibles par la JSP.

Pour tester cette JSP avec Tomcat, il faut compiler le bean Personne dans le répertoire c:\jakarta-tomcat\webapps\examples\web-inf\classes\test et placer le fichier TestBean.jsp dans le répertoire c:\jakarta-tomcat\webapps\examples\jsp\test.

56.4.4.2. Le tag <jsp:setProperty >

Le tag <jsp:setProperty> permet de mettre à jour la valeur d'un ou plusieurs attributs d'un Bean. Le tag utilise le setter (méthode setXXX() ou XXX est le nom de la propriété avec la première lettre en majuscule) pour mettre à jour la valeur. Le bean doit exister grâce à un appel au tag <jsp:useBean>.

Il existe trois façons de mettre à jour les propriétés soit à partir des paramètres de la requête soit avec une valeur :

- alimenter automatiquement toutes les propriétés avec les paramètres correspondants de la requête
- alimenter automatiquement une propriété avec le paramètre de la requête correspondant
- alimenter une propriété avec la valeur précisée

La syntaxe est la suivante :

<jsp:setProperty name="*beanInstanceName*"
{ property="*" |
property="*propertyName*" [param=" *parameterName*"] |
property="*propertyName*" value="{*string* | <%= *expression*%>}"
}
/>

L'attribut name doit contenir le nom de la variable qui contient la référence du bean. Cette valeur doit être identique à celle de l'attribut id du tag <jsp:useBean> utilisé pour instancier le bean.

L'attribut property= «*» permet d'alimenter automatiquement les propriétés du bean avec les paramètres correspondants contenus dans la requête. Le nom des propriétés et le nom des paramètres doivent être identiques.

Comme les paramètres de la requête sont toujours fournis sous forme de String, une conversion est réalisée en utilisant la méthode valueOf() du wrapper du type de la propriété.

Exemple :

```
<jsp:setProperty name="monBean" property="*" />
```

L'attribut property="*propertyName*" [param="*parameterName*"] permet de mettre à jour un attribut du bean. Par défaut, l'alimentation est faite automatiquement avec le paramètre correspondant dans la requête. Si le nom de la propriété et du paramètre sont différents, il faut préciser l'attribut property et l'attribut param qui doit contenir le nom du paramètre qui va alimenter la propriété du bean.

Exemple :

```
<jsp:setProperty name="monBean" property="nom"/>
```

L'attribut property="*propertyName*" value="{*string* | <%= *expression* %>}" permet d'alimenter la propriété du bean avec une valeur particulière.

Exemple :

```
<jsp:setProperty name="monBean" property="nom" value="toto" />
```

Il n'est pas possible d'utiliser param et value dans le même tag.

Exemple : Cette exemple est identique au précédent

```
<html>
<HEAD>
<TITLE>Essai d'instanciation d'un bean dans une JSP</TITLE>
</HEAD>
<body>
<p>Test d'utilisation d'un Bean dans une JSP </p>
<jsp:useBean id="personne" scope="request" class="test.Personne" />
<p>nom initial = <%= personne.getNom() %></p>
<jsp:setProperty name="personne" property="nom" value="mon nom" />
<p>nom mis à jour = <%= personne.getNom() %></p>
</body>
</html>
```

Ce tag peut être utilisé entre les tags <jsp:useBean> et </jsp:useBean> pour initialiser les propriétés du bean lors de son instanciation.

56.4.4.3. Le tag <jsp:getProperty>

Le tag <jsp:getProperty> permet d'obtenir la valeur d'un attribut d'un Bean. Le tag utilise le getter (méthode getXXX() ou XXX est le nom de la propriété avec la première lettre en majuscule) pour obtenir la valeur et l'insérer dans la page HTML généré. Le bean doit exister grâce à un appel au tag <jsp:useBean>.

La syntaxe est la suivante :

<jsp:getProperty name="*beanInstanceName*" property=" *propertyName*" />

L'attribut name indique le nom du bean tel qu'il a été déclaré dans le tag <jsp:useBean>.

L'attribut property indique le nom de la propriété dont on veut la valeur.

Exemple :

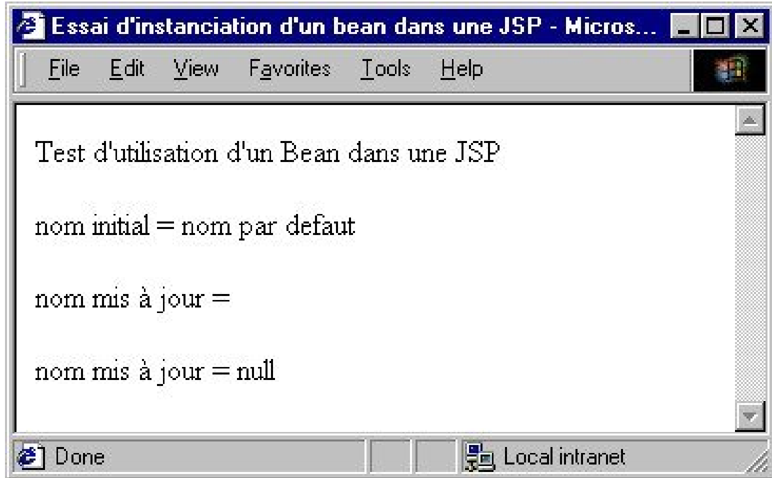
```
<html>
<HEAD>
<TITLE>Essai d'instanciation d'un bean dans une JSP</TITLE>
</HEAD>
<body>
<p>Test d'utilisation d'un Bean dans une JSP </p>
<jsp:useBean id="personne" scope="request" class="test.Personne" />
<p>nom initial = <jsp:getProperty name="personne" property="nom" /></p>
<jsp:setProperty name="personne" property="nom" value="mon nom" />
<p>nom mise à jour = <jsp:getProperty name="personne" property="nom" /></p>
</body>
</html>
```

Attention : ce tag ne permet pas d'obtenir la valeur d'une propriété indexée ni les valeurs d'un attribut d'un EJB.

Remarque : avec Tomcat 3.1, l'utilisation du tag <jsp:getProperty> sur un attribut dont la valeur est null n'affiche rien alors que l'utilisation d'un tag d'expression retourne « null ».

Exemple :

```
<html>
<HEAD>
<TITLE>Essai d'instanciation d'un bean dans une JSP</TITLE>
</HEAD>
<body>
<p>Test d'utilisation d'un Bean dans une JSP </p>
<jsp:useBean id="personne" scope="request" class="test.Personne" />
<p>nom initial = <jsp:getProperty name="personne" property="nom" /></p>
<% personne.setNom(null);%>
<p>nom mis à jour = <jsp:getProperty name="personne" property="nom" /></p>
<p>nom mis à jour = <%= personne.getNom() %></p>
</body>
</html>
```



56.4.4.4. Le tag de redirection <jsp:forward>

Le tag <jsp:forward> permet de rediriger la requête vers une autre URL pointant vers un fichier HTML, JSP ou un servlet.

Dès que le moteur de JSP rencontre ce tag, il redirige le requête vers l'URL précisée et ignore le reste de la JSP courante. Tout ce qui a été généré par la JSP est perdu.

La syntaxe est la suivante :

<jsp:forward page="{*relativeURL* | <%= *expression* %>}" />
ou
<jsp:forward page="{*relativeURL* | <%= *expression* %>}" >
<jsp:param name="*parameterName*" value="{ *parameterValue* | <%= *expression* %>}" /> +
</jsp:forward>

L'option page doit contenir la valeur de l'URL de la ressource vers laquelle la requête va être redirigée.

Cette URL est absolue si elle commence par un '/' sinon elle est relative à la JSP . Dans le cas d'une URL absolue, c'est le serveur web qui détermine la localisation de la ressource.

Il est possible de passer un ou plusieurs paramètres vers la ressource appelée grâce au tag <jsp :param>.

Exemple : Test8.jsp

```
<html>
<body>
<p>Page initiale appelée</p>
<jsp:forward page="forward.htm"/>
</body>
</html>
forward.htm
```

```
<HTML>
<HEAD>
<TITLE>Page HTML</TITLE>
</HEAD>
<BODY>
<p><table border="1" cellpadding="4" cellspacing="0" width="30%" align=center >
<tr bgcolor="#A6A5C2">
<td align="center">Page HTML forwardée</Td>
</Tr>
</table></p>
</BODY>
</HTML>
```

Dans l'exemple, le fichier forward.htm doit être dans le même répertoire que la JSP. Lors de l'appel à la JSP, c'est la page HTML qui est affichée. Le contenu généré par la page JSP n'est pas affiché.

56.4.4.5. Le tag <jsp:include>

Ce tag permet d'inclure le contenu généré par une JSP ou une servlet dynamiquement au moment où la JSP est exécutée. C'est la différence avec la directive include avec laquelle le fichier est inséré dans la JSP avant la génération de la servlet.

La syntaxe est la suivante :

<jsp:include page="*relativeURL" flush="true"* />

L'attribut page permet de préciser l'URL relative de l'élément à insérer.

L'attribut flush permet d'indiquer si le tampon doit être envoyé au client et vidé. Si la valeur de ce paramètre est true, il n'est pas possible d'utiliser certaines fonctionnalités dans la servlet ou la JSP appelée : il n'est pas possible de modifier l'entête de la réponse (header, cookies) ou renvoyer ou faire suivre vers une autre page.

Exemple :

```
<html>
  <body>
    <jsp:include page="bandeau.jsp"/>
    <H1>Bonjour</H1>
    <jsp:include page="pied.jsp"/>
  </body>
</html>
```

Il est possible de fournir des paramètres à la servlet ou à la JSP appelée en utilisant le tag <jsp:param>.

56.4.4.6. Le tag <jsp:plugin>

Ce tag permet la génération du code HTML nécessaire à l'exécution d'une applet en fonction du navigateur : un tag HTML <Object> ou <Embed> est généré en fonction de l'attribut User-Agent de la requête.

Le tag <jsp:plugin> possède trois attributs obligatoires :

Attribut	Rôle
code	permet de préciser le nom de classe
codebase	contient une URL précisant le chemin absolu ou relatif du répertoire contenant la classe ou l'archive
type	les valeurs possibles sont applet ou bean

Il possède aussi plusieurs autres attributs optionnels dont les plus utilisés sont :

Attribut	Rôle
align	permet de préciser l'alignement de l'applet : les valeurs possibles sont bottom, middle ou top
archive	permet de préciser un ensemble de ressources (bibliothèques jar, classes, ...) qui seront automatiquement chargées. Le chemin de ces ressources tient compte de l'attribut codebase
height	précise la hauteur de l'applet en pixel ou en pourcentage
hspace	précise le nombre de pixels insérés à gauche et à droite de l'applet
jrversion	précise la version minimale du jre à utiliser pour faire fonctionner l'applet
name	précise le nom de l'applet
vspace	précise le nombre de pixels insérés en haut et en bas de l'applet
width	précise la longueur de l'applet en pixel ou en pourcentage

Pour fournir un ou plusieurs paramètres, il faut utiliser dans le corps du tag <jsp:plugin> le tag <jsp:params>. Chaque paramètre sera alors défini dans un tag <jsp:param>.

Exemple :

```
<jsp:plugin type="applet" code="MonApplet.class" codebase="applets"
          jreversion="1.1" width="200" height="200" >
  <jsp:params>
     <jsp:param name="couleur" value="eeeeee" />
  </jsp:params>
</jsp:plugin>
```

Le tag <jsp:fallback> dans le corps du tag <jsp:plugin> permet de préciser un message qui sera affiché dans les navigateurs ne supportant pas le tag HTML <Object> ou <Embed>.

56.5. Un Exemple très simple

Exemple : TestJSPIdent.html

```
<HTML>
<HEAD>
<TITLE>Identification</TITLE>
</HEAD>
<BODY>
<FORM METHOD=POST ACTION="jsp/TestJSPAccueil.jsp">
Entrer votre nom :
<INPUT TYPE=TEXT NAME="nom">
<INPUT TYPE=SUBMIT VALUE="SUBMIT">
</FORM>
</BODY>
</HTML>
```

Exemple : TestJSPAccueil.jsp

```
<HTML>
<HEAD>
<TITLE>Accueil</TITLE>
</HEAD>
<BODY>
<%
String nom = request.getParameter("nom");
%>
<H2>Bonjour <%= nom %></H2>
```

```
</BODY>
</HTML>
```

56.6. La gestion des erreurs

Lors de l'exécution d'une page JSP, des erreurs peuvent survenir. Chaque erreur se traduit par la levée d'une exception. Si cette exception est capturée dans un bloc try/catch de la JSP, celle ci est traitée. Si l'exception n'est pas capturée dans la page, il y a deux possibilités selon qu'une page d'erreur soit associée à la page JSP :

- sans page d'erreur associée, la pile d'exécution de l'exception est affichée
- avec une page d'erreur associée, une redirection est effectuée vers cette JSP

La définition d'une page d'erreur permet de la préciser dans l'attribut errorPage de la directive page des autres JSP de l'application. Si une exception est levée dans les traitements d'une de ces pages, la JSP va automatiquement rediriger l'utilisateur va la page d'erreur précisée.

La valeur de l'attribut errorPage de la directive page doit contenir l'URL de la page d'erreur. Le plus simple est de définir cette page à la racine de l'application web et de faire précéder le nom de la page par un caractère '/' dans l'url.

Exemple :

```
<%@ page errorPage="/mapagederreur.jsp" %>
```

56.6.1. La définition d'une page d'erreur

Une page d'erreur est une JSP dont l'attribut isErrorPage est égal à true dans la directive page. Une telle page dispose d'un accès à la variable implicite nommée exception de type Throwable qui encapsule l'exception qui a été levée.

Il est possible dans une telle page d'afficher un message d'erreur personnalisé mais aussi d'inclure des traitements liés à la gestion de l'exception : ajouter l'exception dans un journal, envoie d'un mail pour son traitement, ...

Exemple :

```
<%@ page language="java" contentType="text/html" %>
%@ page isErrorPage="true" %>
<html>
  <body>
    <h1>Une erreur est survenue lors des traitements</h1>
    <p><%= exception.getMessage() %></p>
</body>
</html>
```

56.7. Les bibliothèques de tags personnalisés (custom taglibs)

Les bibliothèques de tags (taglibs) ou tags personnalisés (custom tags) permettent de définir ses propres tags basés sur XML, de les regrouper dans une bibliothèque et de les réutiliser dans des JSP. C'est une extension de la technologie JSP apparue à partir de la version 1.1 des spécifications des JSP.

56.7.1. La présentation des tags personnalisés

Un tag personnalisé est un élément du langage JSP défini par un développeur pour des besoins particuliers qui ne sont pas traités en standard par les JSP. Elles permettent de définir ces propres tags qui réaliseront des actions pour générer la réponse.

Le principal but est de favoriser la séparation des rôles entre le développeur Java et concepteur de page web. L'idée maitresse est de déporter le code Java contenu dans les scriplets de la JSP dans des classes dédiées et de les appeler dans le code source de la JSP en utilisant des tags particuliers.

Ce concept peut sembler proche de celui des javabeans dont le rôle principal est aussi de définir des composants réutilisables. Les javabeans sont particulièrement adaptés pour stocker et échanger des données entre les composants de l'application web via la session.

Les tags personnalisés sont adaptés pour enlever du code Java inclus dans les JSP est le déporter dans une classe dédiée. Cette classe est physiquement un javabean qui implémente une interface particulière.

La principale différence entre un javabean et un tag personnalisé est que ce dernier tient compte de l'environnement dans lequel il s'exécute (notamment la JSP et le contexte de l'application web) et interagit avec lui.

Pour de plus amples informations sur les bibliothèques de tags personnalisés, il suffit de consulter le site de Sun qui leur sont consacrées :
http://java.sun.com/products/jsp/taglibraries.html.

Les tags personnalisés possèdent des fonctionnalités intéressantes :

- ils ont un accès aux objets de la JSP notamment l'objet de type HttpResponse. Ils peuvent donc modifier le contenu de la réponse générée par la JSP
- ils peuvent recevoir des paramètres envoyés à partir de la JSP qui les appelle
- ils peuvent avoir un corps qu'ils peuvent manipuler. Par extension de cette fonctionnalité, il est possible d'imbriquer un tag personnalisé dans un autre avec un nombre d'imbrication illimité

Les avantages des bibliothèques de tags personnalisés sont :

- une suppression du code Java dans la JSP remplacé par un tag XML facilement compréhensible ce qui simplifie grandement la JSP
- une API facile à mettre en oeuvre
- une forte et facile réutilisabilité des tags développés
- une maintenance des JSP facilitée

La définition d'une bibliothèque de tags comprend plusieurs entités :

- une classe dit "handler" pour chaque tag qui compose la bibliothèque
- un fichier de description de la bibliothèque

56.7.2. Les handlers de tags

Chaque tag est associé à une classe qui va contenir les traitements à exécuter lors de l'utilisation du tag. Une telle classe est nommée "handler de tag" (tag handler). Pour permettre leur appel, une telle classe doit obligatoirement implémenter directement ou indirectement l'interface javax.servlet.jsp.tagext.Tag

L'interface Tag possède une interface fille BodyTag qui doit être utilisée dans le cas ou le tag peut utiliser le contenu de son corps.

Pour plus de facilité, l'API JSP propose les classes TagSupport et BodyTagSupport qui implémentent respectivement l'interface Tag et BodyTag. Ces deux classes, contenues dans le package javax.servlet.jsp.tagext, proposent des implémentations par défaut des méthodes de l'interface. Ces deux classes proposent un traitement standard par défaut pour chacune des méthodes de l'interface qu'ils implémentent. Pour définir un handler de tag, il suffit d'hériter de l'une ou l'autre de ces deux classes.

Les méthodes définies dans les interfaces Tag et BodyTag sont appelées, par la servlet issue de la compilation de la JSP, au cours de l'utilisation du tag.

Le cycle de vie général d'un tag est le suivant :

- lors de la rencontre du début du tag, un objet du type du handler est instancié
- plusieurs propriétés sont initialisées (pageContext, parent, ...) en utilisant les setters correspondant
- si le tag contient des attributs, les setters correspondant sont appelés pour alimenter leur valeur
- la méthode doStartTag() est appelée
- si la méthode doStartTag() renvoie la valeur EVAL_BODYINCLUDE alors le contenu du corps du tag est évalué
- lors de la rencontre de la fin du tag, appel de la méthode doEndTag()
- si la méthode doEndTag() renvoie la valeur EVAL_PAGE alors l'évaluation de la page se poursuit, si elle renvoie la valeur SKIP_PAGE elle ne se poursuit pas

Toutes ces opérations sont réalisées par le code généré lors de la compilation de la JSP.

Un handler de tag possède un objet qui permet d'avoir un accès aux objets implicites de la JSP. Cet objet est du type javax.servlet.jsp.PageContext

Comme le code contenu dans la classe du tag ne peut être utilisé que dans le contexte particulier du tag, il peut être intéressant de sortir une partie de ce code dans une ou plusieurs classes dédiées qui peuvent être éventuellement des beans.

Pour compiler ces classes, il faut obligatoirement que le jar de l'API servlets (servlets.jar) soit inclus dans la variable CLASSPATH.

56.7.3. L'interface Tag

Cette interface définit les méthodes principales pour la gestion du cycle de vie d'un tag personnalisé qui ne doit pas manipuler le contenu de son corps.

Elle définit plusieurs constantes :

Constante	Rôle
EVAL_BODY_INCLUDE	Continuer avec l'évaluation du corps du tag
EVAL_PAGE	Continuer l'évaluation de la page
SKIP_BODY	Empêcher l'évaluation du corps du tag
SKIP_PAGE	Empêcher l'évaluation du reste de la page

Elle définit aussi plusieurs méthodes :

Méthode	Rôle
int doEndTag()	Traitements à la rencontre du tag de début
int doStartTag()	Traitements à la rencontre du tag de fin
setPageContext(Context)	Sauvegarde du contexte de la page

La méthode doStartTag() est appelée lors de la rencontre du tag d'ouverture et contient les traitements à effectuer dans ce cas. Elle doit renvoyer un entier prédéfini qui indique comment va se poursuivre le traitement du tag :

- EVAL_BODY_INCLUDE : poursuite du traitement avec évaluation du corps du tag
- SKIP_BODY : poursuite du traitement sans évaluation du corps du tag

La méthode doEndTag() est appelée lors de la rencontre du tag de fermeture et contient les traitements à effectuer dans ce cas. Elle doit renvoyer un entier prédéfini qui indique comment va se poursuivre le traitement de la JSP.

* EVAL_PAGE : poursuite du traitement de la JSP
* SKIP_PAGE : ne pas poursuivre le traitement du reste de la JSP

56.7.4. L'accès aux variables implicites de la JSP

Les tags ont accès aux variables implicites de la JSP dans laquelle ils s'exécutent via un objet de type PageContext. La variable pageContext est un objet de ce type qui est initialisé juste après l'instanciation du handler.

Le classe PageContext est une classe abstraite dont l'implémentation des spécifications doit fournir une adaptation concrète.

Cette classe définit plusieurs méthodes :

Méthodes	Rôles
JspWriter getOut()	Permet un accès à la variable out de la JSP
Exception getException()	Permet un accès à la variable exception de la JSP
Object getPage()	Permet un accès à la variable page de la JSP
ServletRequest getRequest()	Permet un accès à la variable request de la JSP
ServletResponse getResponse()	Permet un accès à la variable response de la JSP
ServletConfig getServletConfig()	Permet un accès à l'instance de la variable de type ServletConfig
ServletContext getServletContext()	Permet un accès à l'instance de la variable de type ServletContext
HttpSession getSession()	Permet un accès à la session
Object getAttribute(String)	Renvoie l'objet associé au nom fourni en paramètre dans la portée de la page
setAttribute(String, Object)	Permet de placer dans la portée de la page un objet dont le nom est fourni en paramètre

56.7.5. Les deux types de handlers

Il existe deux types de handlers :

* les handlers de tags sans corps
* les handlers de tags avec corps

56.7.5.1. Les handlers de tags sans corps

Pour définir le handler d'un tag personnalisé sans corps, il suffit de définir une classe qui implémente l'interface Tag ou qui héritent de la classe TagSupport. Il faut définir ou redéfinir les méthodes doStartTag() et endStartTag()

La méthode doStartTag() est appelée à la rencontre du début du tag. Cette méthode doit contenir le code à exécuter dans ce cas et renvoyer la constante SKIP_BODY puisque le tag ne contient pas de corps

56.7.5.2. Les handlers de tags avec corps

Le cycle de vie d'un tel tag inclus le traitement du corps si la méthode doStartTag() renvoie la valeur EVAL_BODY_TAG.

Dans ce cas, les opérations suivantes sont réalisées :

- la méthode setBodyContent() est appelée
- le contenu du corps est traité
- la méthode doAfterBody() est appelée. Si elle renvoie la valeur EVAL_BODY_TAG, le contenu du corps est de nouveau traité

56.7.6. Les paramètres d'un tag

Un tag peut avoir un ou plusieurs paramètres qui seront transmis à la classe via des attributs. Pour chacun des paramètres, il faut définir des getter et des setter en respectant les règles et conventions des Java beans. Il est impératif de définir un champ, un setter et éventuellement un accesseur pour chaque attribut.

La JSP utilisera le setter pour fournir à l'objet la valeur de l'attribut.

Au moment de la génération de la servlet par le moteur de JSP, celui ci vérifie par introspection la présence d'un setter pour l'attribut concerné.

56.7.7. La définition du fichier de description de la bibliothèque de tags (TLD)

Le fichier de description de la bibliothèque de tags (tag library descriptor file) est un fichier au format XML qui décrit une bibliothèque de tag. Les informations qu'il contient concerne la bibliothèque de tags elle même et concerne aussi chacun des tags qui la compose.

Ce fichier est utilisé par le conteneur Web lors de la compilation de la JSP pour remplacer le tag par du code Java.

Ce fichier doit toujours avoir pour extension .tld. Il doit être placé dans le répertoire web-inf du fichier war ou dans un de ces sous répertoires. Le plus pratique est de tous les regrouper dans un répertoire nommé par exemple tags ou tld.

Comme tout bon fichier XML, le fichier TLD commence par un prologue :

Exemple :

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
  PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
  "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">
```

La DTD précisée doit correspondre à la version de l'API JSP utilisée. L'exemple précédent concernait la version 1.1, l'exemple suivant concerne la version 1.2

Exemple :

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
  PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
  "http://java.sun.com/dtds/web-jsptaglibrary_1_2.dtd">
```

Le tag racine du document XML est le tag <taglib>.

Ce tag peut contenir plusieurs tags qui définissent les caractéristiques générales de la bibliothèque. Les tags suivants sont

définis dans les spécifications 1.2 :

Nom	Rôle
tlib-version	version de la bibliothèque
jsp-version	version des spécifications JSP utilisée
short-name	nom court la bibliothèque (optionnel)
uri	URI qui identifie de façon unique la bibliothèque : cette URI n'a pas besoin d'exister réellement
display-name	nom de la bibliothèque
small-icon	(optionnel)
large-icon	(optionnel)
description	description de la bibliothèque
validator	(optionnel)
listener	(optionnel)
tag	il en faut autant que de tags qui composent la bibliothèque

Pour chaque tag personnalisé défini dans la bibliothèque, il faut un tag <tag>. Ce tag permet de définir les caractéristiques d'un tag de la bibliothèque.

Ce tag peut contenir les tags suivants :

Nom	Rôle
name	nom du tag : il doit être unique dans la bibliothèque
tag-class	nom entièrement qualifié de la classe qui contient le handler du tag
tei-class	nom qualifié d'une classe fille de la classe javax.servlet.jsp.tagext.TagExtraInfo (optionnel)
body-content	type du corps du tag. Les valeurs possibles sont : • JSP : le corps du tag contient des tags JSP qui doivent être interprétés • tagdependent : l'interprétation du contenu du corps est faite par le tag • empty : le corps doit obligatoirement être vide La valeur par défaut est JSP
display-name	nom court du tag
small-icon	nom relatif par rapport à la bibliothèque d'un fichier gif ou jpeg contenant une icône. (optionnel)
large-icon	nom relatif par rapport à la bibliothèque d'un fichier gif ou jpeg contenant une icône. (optionnel)
description	description du tag (optionnel)
variable	(optionnel)
attribute	il en faut autant que d'attribut possédé par le tag (optionnel)
example	un exemple de l'utilisation du tag (optionnel)

Pour chaque attribut du tag personnalisé, il faut utiliser un tag <attribute>. Ce tag décrit un attribut d'un tag et peut contenir les tags suivants :

Nom	Description
name	nom de l'attribut

required	booléen qui indique la présence obligatoire de l'attribut
rtexprvalue	booléen qui indique si la page doit évaluer l'expression lors de l'exécution. Il faut donc mettre la valeur true si la valeur de l'attribut est fournie avec un tag JSP d'expression <%= %>

Le tag <Variable> contient les tags suivants :

Nom	Rôle
name-given	
name-from-attribut	
variable-class	nom de la classe de la valeur de l'attribut. Par défaut java.lang.String
declare	par défaut : True
scope	visibilité de l'attribut. Les valeurs possibles sont : • AT_BEGIN • NESTED • AT_END Par défaut : NESTED (optionnel)
description	description de l'attribut (optionnel)

Chaque bibliothèque doit être définie avec un fichier de description au format xml possédant une extension .tld. Le contenu de ce fichier doit pouvoir être validé avec une DTD fournie par Sun.

Ce fichier est habituellement stocké dans le répertoire web-inf de l'application web ou un de ses sous répertoires.

Exemple :

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
  <!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
  "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">
<taglib>
  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>testtaglib</shortname>
  <uri>http://perso.jmd.test.taglib</uri>
  <info>Bibliotheque de test des taglibs</info>

  <tag>
  <name>testtaglib1</name>
  <tagclass>perso.jmd.test.taglib.TestTaglib1</tagclass>
  <info>Tag qui affiche bonjour</info>
  </tag>
</taglib>
```

56.7.8. L'utilisation d'une bibliothèque de tags

Pour utiliser une bibliothèque de classe, il y a des actions à réaliser au niveau du code source de la JSP et au niveau de conteneur d'application web pour déployer la bibliothèque de tags.

56.7.8.1. L'utilisation dans le code source d'une JSP

Pour chaque bibliothèque à utiliser dans une JSP, il faut la déclarer en utilisant la directive taglib avant son utilisation. Le plus simple est d'effectuer ces déclarations tout au début du code de la JSP.

Cette directive possède deux attributs :

- uri : l'URI de la bibliothèque telle que définie dans le fichier de description
- prefix : un préfix qui servira d'espace de noms pour les tags de la bibliothèque dans la JSP

Exemple :

```
<%@ taglib uri="/WEB-INF/tld/testtaglib.tld" prefix="maTagLib" %>
```

L'attribut uri permet de donner une identité au fichier de description de la bibliothèque de tags (TLD). La valeur fournie peut être :

- directe (par exemple le nom du fichier avec son chemin relatif)

Exemple :

```
<%@ taglib uri="/WEB-INF/tld/testtaglib.tld" prefix="maTagLib" %>
```

- ou indirecte (concordance avec un nom logique défini dans un tag taglib du descripteur de déploiement de l'application web)

Exemple :

```
<%@ taglib uri= "/maTaglib" prefix= "maTagbib" %>
```

Dans ce dernier cas, il faut ajouter pour chaque bibliothèque un tag <taglib> dans le fichier de description de déploiement de l'application/WEB-INF/web.xml

Exemple :

```
  <taglib>
    <taglib-uri>/maTagLibTest</taglib-uri>
    <taglib-location>/WEB-INF/tld/testtaglib.tld</taglib-location>
  </taglib>
```

L'appel d'un tag se fait en utilisant un tag dont le nom à la forme suivante : prefix:tag

Le préfix est celui défini dans la directive taglib.

Exemple : un tag sans corps

```
<maTagLib:testtaglib1/>
```

Exemple : un tag avec corps

```
<prefix:tag>
   ...
</prefix:tag>
```

Le corps peut contenir du code HTML, du code JSP ou d'autre tag personnalisé.

Le tag peut avoir des attributs si ceux ci ont été définis. La syntaxe pour les utiliser respecte la norme XML

Exemple : un tag avec un paramètre constant

```
<prefix:tag attribut="valeur"/>
```

La valeur de cet attribut peut être une donnée dynamiquement évaluée lors de l'exécution :

Exemple : un tag avec un paramètre

```
<prefix:tag attribut="<%= uneVariable%>"/>
```

56.7.8.2. Le déploiement d'une bibliothèque

Au moment de la compilation de la JSP en servlet, le conteneur transforme chaque tag en un appel à un objet du type de la classe associé au tag.

Il y a deux types d'éléments dont il faut s'assurer l'accès par le conteneur d'applications web :

- le fichier de description de la bibliothèque
- les classes des handlers de tag

Les classes des handlers de tags peuvent être stockées à deux endroits dans le fichier war selon leur format :

- si ils sont packagés sous forme de fichier jar alors ils doivent être placés dans le répertoire /WEB-INF/lib
- si ils ne sont pas packagés alors ils doivent être placés dans le répertoire /WEB-INF/classes

56.7.9. Le déploiement et les tests dans Tomcat

Tomcat étant l'implémentation de référence pour les technologies servlets et JSP, il est pratique d'effectuer des tests avec cet outil.

La version de Tomcat utilisée dans cette section est la 3.2.1.

Le déploiement se fait en deux étapes :

- la copie des fichiers
- L'enregistrement de la bibliothèque

Les classes compilées doivent être copiées dans le répertoire WEB-INF/classes de la webapp si elles ne sont pas packagées dans une archive jar, sinon le ou les fichiers .jar doivent être copiés dans le répertoire WEB-INF/lib.

Le fichier .tld doit être copié dans le répertoire WEB-INF ou dans un de ces sous répertoires.

Il faut ensuite enregistrer la bibliothèque dans le fichier de configuration web.xml contenu dans le répertoire web-inf du répertoire de l'application web.

Il faut ajouter dans ce fichier, un tag <taglib> pour chaque bibliothèque utilisée par l'application web contenant deux informations :

- l'URI de la bibliothèque contenue dans le tag taglib-uri. Cette URI doit être identique à celle définie dans le fichier de description de la bibliothèque
- la localisation du fichier de description

Exemple :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

```
<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
  "http://java.sun.com/j2ee/dtds/web-app_2.2.dtd">
<web-app>
  <welcome-file-list id="ListePageDAccueil">
    <welcome-file>index.htm</welcome-file>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>

  <taglib>
    <taglib-uri>/maTagLibTest</taglib-uri>
    <taglib-location>/WEB-INF/tld/testtaglib.tld</taglib-location>
  </taglib>
</web-app>
```

Il ne reste plus qu'à lancer Tomcat si ce n'est pas encore fait et de saisir l'url de la page contenant l'appel au tag personnalisé.

56.7.10. Les bibliothèques de tags existantes

Il existe de nombreuses bibliothèques de tags libres ou commerciales disponibles sur le marché. Cette section va tenter de présenter quelques unes de plus connues et des plus utilisées du monde libre. Cette liste n'est pas exhaustive.

56.7.10.1. Struts

Struts est un framework pour la réalisation d'applications web reposant sur le modèle MVC 2.

Pour la partie vue, Struts utilise les JSP et propose en plus plusieurs bibliothèques de tags pour faciliter le développement de cette partie présentation. Struts possède quatre grandes bibliothèques :

- formulaire HMTL
- modèles (templates)
- Javabeans (bean)
- traitements logiques (logic)

Le site web de struts se trouve à l'url : http://jakarta.apache.org/struts/index.html

Ce framework est détaillée dans le chapitre «Struts».

56.7.10.2. Jakarta Tag libs

La suite de ce chapitre sera développée dans une version future de ce document

56.7.10.3. JSP Standard Tag Library (JSTL)

JSP Standard Tag Library (JSTL) est une spécification issu du travail du JCP sous la JSR numéro 52. Le chapitre «JSTL (Java server page Standard Tag Library)» fournit plus de détails sur cette spécification.

57. JSTL (Java server page Standard Tag Library)

JSTL est l'acronyme de Java server page Standard Tag Library. C'est un ensemble de tags personnalisés développé sous la JSR 052 qui propose des fonctionnalités souvent rencontrées dans les JSP :

- Tag de structure (itération, conditionnement ...)
- Internationalisation
- Exécution de requête SQL
- Utilisation de document XML

JSTL nécessite un conteneur d'application web qui implémente l'API servlet 2.3 et l'API JSP 1.2. L'implémentation de référence (JSTL-RI) de cette spécification est développée par le projet Taglibs du groupe Apache sous le nom " Standard ".

Il est possible de télécharger cette implémentation de référence à l'URL :
http://jakarta.apache.org/taglibs/doc/standard-doc/intro.html

JSTL est aussi inclus dans le JWSDP (Java Web Services Developer Pack), ce qui facilite son installation et son utilisation. Les exemples de cette section ont été réalisés avec le JWSDP 1.001

JSTL possède quatre bibliothèques de tag :

Rôle	TLD	Uri
Fonctions de base	c.tld	http://java.sun.com/jstl/core
Traitements XML	x.tld	http://java.sun.com/jstl/xml
Internationalisation	fmt.tld	http://java.sun.com/jstl/fmt
Traitements SQL	sql.tld	http://java.sun.com/jstl/sql

JSTL propose un langage nommé EL (expression langage) qui permet de faire facilement référence à des objets java accessibles dans les différents contextes de la JSP.

La bibliothèque de tag JSTL est livrée en deux versions :

- JSTL-RT : les expressions pour désigner des variables utilisant la syntaxe JSP classique
- JSTL-EL : les expressions pour désigner des variables utilisant le langage EL

Pour plus informations, il est possible de consulter les spécifications à l'url suivante :
http://jcp.org/aboutJava/communityprocess/final/jsr052/

Ce chapitre contient plusieurs sections :

- Un exemple simple
- Le langage EL (Expression Langage)
- La bibliothèque Core
- La bibliothèque XML
- La bibliothèque I18n

♦ La bibliothèque Database

57.1. Un exemple simple

Pour commencer, voici un exemple et sa mise en oeuvre détaillée. L'application web d'exemple se nomme test. Il faut créer un répertoire test dans le répertoire webapps de tomcat.

Pour utiliser JSTL, il faut copier les fichiers jstl.jar et standard.jar dans le répertoire WEB-INF/lib de l'application web.

Il faut copier les fichiers .tld dans le répertoire WEB-INF ou un de ses sous répertoires. Dans la suite de l'exemple, ces fichiers ont été placés dans le répertoire /WEB-INF/tld.

Il faut ensuite déclarer les bibliothèques à utiliser dans le fichier web.xml du répertoire WEB-INF comme pour toute bibliothèque de tags personnalisés.

Exemple : pour la bibliothèque Core

```
<taglib>
    <taglib-uri>http://java.sun.com/jstl/core</taglib-uri>
<taglib-location>/WEB-INF/tld/c.tld</taglib-location>
</taglib>
```

L'arborescence des fichiers est la suivante :

Exemple :

```
webapps
      test
            WEB-INF
                  lib
                        jstl.jar
                        standard.jar
                  tld
                        c.tld
                  web.xml
            test.jsp
```

Pour pouvoir utiliser une bibliothèque personnalisée, il faut utiliser la directive taglib :

Exemple :

```
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
```

Voici les codes source des différents fichiers de l'application web :

Exemple : fichier test.jsp

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<html>
  <head>
    <title>Exemple</title>
  </head>

  <body>
      <c:out value="Bonjour" /><br/>
  </body>
</html>
```

Exemple : le fichier WEB-INF/web.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app23.dtd">

<web-app>
  <taglib>
    <taglib-uri>http://java.sun.com/jstl/core</taglib-uri>
<taglib-location>/WEB-INF/tld/c.tld</taglib-location>
</taglib>
</web-app>
```

Pour tester l'application, il suffit de lancer Tomcat et de saisir l'url localhost:8080/test/test.jsp dans un browser.

57.2. Le langage EL (Expression Langage)

JSTL propose un langage particulier constitué d'expressions qui permet d'utiliser et de faire référence à des objets java accessible dans les différents contextes de la page JSP. Le but est de fournir un moyen simple d'accéder aux données nécessaires à une JSP.

La syntaxe de base est ${xxx} ou xxx est le nom d'une variable d'un objet java défini dans un contexte particulier. La définition dans un contexte permet de définir la portée de la variable (page, requête, session ou application).

EL permet facilement de s'affranchir de la syntaxe de java pour obtenir une variable.

Exemple : accéder à l'attribut nom d'un objet personne situé dans la session avec Java

```
<%= session.getAttribute("personne").getNom() %>
```

Exemple : accéder à l'attribut nom d'un objet personne situé dans la session avec EL

```
${sessionScope.personne.nom}
```

EL possède par défaut les variables suivantes :

Variable	Rôle
PageScope	variable contenue dans la portée de la page (PageContext)
RequestScope	variable contenue dans la portée de la requête (HttpServletRequest)
SessionScope	variable contenue dans la portée de la session (HttpSession)
ApplicationScope	variable contenue dans la portée de l'application (ServletContext)
Param	paramètre de la requête http
ParamValues	paramètres de la requête sous la forme d'une collection
Header	en tête de la requête
HeaderValues	en têtes de la requête sous la forme d'une collection
InitParam	paramètre d'initialisation
Cookie	cookie
PageContext	objet PageContext de la page

EL propose aussi différents opérateurs :

Operateur	Rôle	Exemple
.	Obtenir une propriété d'un objet	${param.nom}
[]	Obtenir une propriété par son nom ou son indice	${param[" nom "]} ${row[1]}
Empty	Teste si un objet est null ou vide si c'est une chaîne de caractère. Renvoie un booléen	${empty param.nom}
== eq	test l'égalité de deux objet	
!= ne	test l'inégalité de deux objet	
< lt	test strictement inférieur	
> gt	test strictement supérieur	
<= le	test inférieur ou égal	
>= ge	test supérieur ou égal	
+	Addition	
-	Soustraction	
*	Multiplication	
/ div	Division	
% mod	Modulo	
&& and		
|| or		
! not	Négation d'une valeur	

EL ne permet pas l'accès aux variables locales. Pour pouvoir accéder à de telles variables, il faut obligatoirement en créer une copie dans une des portées particulières : page, request, session ou application

Exemple :

```
<%
  int valeur = 101;
%>
  valeur = <c:out value="${valeur}" /><BR/>
```

Résultat :

```
valeur =
```

Exemple : avec la variable copiée dans le contexte de la page

```
<%
  int valeur = 101;
  pageContext.setAttribute("valeur", new Integer(valeur));
```

```
%>
  valeur = <c:out value="${valeur}" /><BR/>
```

Résultat :

```
valeur = 101
```

57.3. La bibliothèque Core

Elle propose les tags suivants répartis dans trois catégories :

Catégorie	Tag
Utilisation de EL	set out remove catch
Gestion du flux (condition et itération)	if choose forEach forTokens
Gestion des URL	import url redirect

Pour utiliser cette bibliothèque, il faut la déclarer dans le fichier web.xml du répertoire WEB-INF de l'application web.

Exemple :

```
<taglib>
    <taglib-uri>http://java.sun.com/jstl/core</taglib-uri>
    <taglib-location>/WEB-INF/tld/c.tld</taglib-location>
</taglib>
```

Dans chaque JSP qui utilise un ou plusieurs tags de la bibliothèque, il faut la déclarer avec une directive taglib

Exemple :

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
```

57.3.1. Le tag set

Le tag set permet de stocker une variable dans une portée particulière (page, requête, session ou application).

Il possède plusieurs attributs :

Attribut	Rôle
value	valeur à stocker
target	nom de la variable contenant un bean dont la propriété doit être modifiée
property	nom de la propriété à modifier

var	nom de la variable qui va stocker la valeur
scope	portée de la variable qui va stocker la valeur

Exemple :

```
    <c:set var="maVariable1" value="valeur1" scope="page" />
    <c:set var="maVariable2" value="valeur2" scope="request" />
    <c:set var="maVariable3" value="valeur3" scope="session" />
    <c:set var="maVariable4" value="valeur4" scope="application" />
```

La valeur peut être déterminée dynamiquement.

Exemple :

```
    <c:set var="maVariable" value="${param.id}" scope="page" />
```

L'attribut target avec l'attribut property permet de modifier la valeur d'une propriété (précisée avec l'attribut property) d'un objet (précisé avec l'attribut target).

La valeur de la variable peut être précisée dans le corps du tag plutôt que d'utiliser l'attribut value.

Exemple :

```
<c:set var="maVariable" scope="page">
       Valeur de ma variable
</c:set>
```

57.3.2. Le tag out

Le tag out permet d'envoyer dans le flux de sortie de la JSP le résultat de l'évaluation de l'expression fournie dans le paramètre " value ". Ce tag est équivalent au tag d'expression <%= ... %> de JSP.

Il possède plusieurs attributs :

Attribut	Rôle
value	valeur à afficher (obligatoire)
default	définir une valeur par défaut si la valeur est null
escapeXml	booléen qui précise si les caractères particuliers (< > & ...) doivent être convertis en leur équivalent HTML (< > & ; ...)

Exemple :

```
    <c:out value='${pageScope.maVariable1}' />
    <c:out value='${requestScope.maVariable2}' />
    <c:out value='${sessionScope.maVariable 3}' />
    <c:out value='${applicationScope.maVariable 4}' />
```

Il n'est pas obligatoire de préciser la portée dans laquelle la variable est stockée : dans ce cas, la variable est recherchée prioritairement dans la page, la requête, la session et enfin l'application.

L'attribut default permet de définir une valeur par défaut si le résultat de l'évaluation de la valeur est null. Si la valeur est null et que l'attribut default n'est pas utilisé alors c'est une chaîne vide qui est envoyée dans le flux de sortie.

Exemple :

```
   <c:out value="${personne.nom}" default="Inconnu" />
```

Le tag out est particulièrement utile pour générer le code dans un formulaire en remplaçant avantageusement les scriplets.

Exemple :

```
<input type="text" name="nom" value="<c:out value="${param.nom}"/>" />
```

57.3.3. Le tag remove

Le tag remove permet de supprimer une variable d'une portée particulière.

Il possède plusieurs attributs :

Attribut	Rôle
var	nom de la variable à supprimer (obligatoire)
scope	portée de la variable

Exemple :

```
    <c:remove var="maVariable1" scope="page" />
    <c:remove var="maVariable2" scope="request" />
    <c:remove var="maVariable3" scope="session" />
    <c:remove var="maVariable4" scope="application" />
```

57.3.4. Le tag catch

Ce tag permet de capturer des exceptions qui sont levées lors de l'exécution du code inclus dans son corps.

Il possède un attribut :

Attribut	Rôle
var	nom d'une variable qui va contenir des informations sur l'anomalie

Si l'attribut var n'est pas utilisé, alors toutes les exceptions levées lors de l'exécution du corps du tag sont ignorées.

Exemple : code non protégé

```
<c:set var="valeur" value="abc" />
<fmt:parseNumber var="valeurInt" value="${valeur}"/>
```

Résultat : une exception est lévée

```
javax.servlet.ServletException: In <parseNumber>, value attribute can not be parsed: "abc"
        at org.apache.jasper.runtime.PageContextImpl.handlePageException(PageContextImpl.java:
471)
        at org.apache.jsp.test$jsp.jspService(test$jsp.java:1187)
        at org.apache.jasper.runtime.HttpJspBase.service(HttpJspBase.java:107)
```

L'utilisation du tag catch peut empêcher le plantage de l'application.

Exemple :

```
<c:set var="valeur" value="abc" />
<c:catch var="erreur">
   <fmt:parseNumber var="valeurInt" value="${valeur}"/>
</c:catch>
<c:if test="${not empty erreur}">
   la valeur n'est pas numerique
</c:if>
```

Résultat :

```
la valeur n'est pas numerique
```

L'objet désigné par l'attribut var du tag catch possède une propriété message qui contient le message d'erreur

Exemple :

```
<c:set var="valeur" value="abc" />
<c:catch var="erreur">
   <fmt:parseNumber var="valeurInt" value="${valeur}"/>
</c:catch>
<c:if test="${not empty erreur}">
 <c:out value="${erreur.message}"/>
</c:if>
```

Résultat :

```
In &lt;parseNumber&gt;, value attribute can not be parsed: "abc"
```

Le souci avec ce tag est qu'il n'est pas possible de savoir quelle exception a été levée.

57.3.5. Le tag if

Ce tag permet d'évaluer le contenu de son corps si la condition qui lui est fournie est vraie.

Il possède plusieurs attributs :

Attribut	Rôle
test	condition à évaluer
var	nom de la variable qui contiendra le résultat de l'évaluation
scope	portée de la variable qui contiendra le résultat

Exemple :

```
<c:if test="${empty personne.nom}" >Inconnu</c:if>
```

Le tag peut ne pas avoir de corps si le tag est simplement utilisé pour stocker le résultat de l'évaluation de la condition dans une variable.

Exemple :

```
<c:if test="${empty personne.nom}" var="resultat" />
```

Le tag if est particulièrement utile pour générer le code dans un formulaire en remplaçant avantageusement les scriplets.

Exemple : selection de la bonne occurrence dont la valeur est fournie en paramètre de la requête

```
<FORM NAME="form1" METHOD="post" ACTION="">
  <SELECT NAME="select">
    <OPTION VALUE="choix1" <c:if test="${param.select == 'choix1'}" >selected</c:if> >
      choix 1</OPTION>
    <OPTION VALUE="choix2" <c:if test="${param.select == 'choix2'}" >selected</c:if> >
      choix 2</OPTION>
    <OPTION VALUE="choix3" <c:if test="${param.select == 'choix3'}" >selected</c:if> >
      choix 3</OPTION>
  </SELECT>
</FORM>
```

Pour tester le code, il faut fournir en paramètre dans l'url select=choix2

Exemple :

```
http://localhost:8080/test/test.jsp?select=choix2
```

57.3.6. Le tag choose

Ce tag permet de traiter différents cas mutuellement exclusifs dans un même tag. Le tag choose ne possède pas d'attribut. Il doit cependant posséder un ou plusieurs tags fils « when ».

Le tag when possède l'attribut test qui permet de préciser la condition à évaluer. Si la condition est vraie alors le corps du tag when est évaluer et le résultat est envoyé dans le flux de sortie de la JSP

Le tag otherwise permet de définir un cas qui ne correspond à aucun des autres inclus dans le tag. Ce tag ne possède aucun attribut.

Exemple :

```
<c:choose>
  <c:when test="${personne.civilite == 'Mr'}">
    Bonjour Monsieur
  </c:when>
  <c:when test="${personne.civilite == 'Mme'}">
    Bonjour Madame
  </c:when>
  <c:when test="${personne.civilite == 'Mlle'}">
    Bonjour Mademoiselle
  </c:when>
  <c:otherwise>
    Bonjour
  </c:otherwise>
</c:choose>
```

57.3.7. Le tag forEach

Ce tag permet de parcourir les différents éléments d'une collection et ainsi d'exécuter de façon répétitive le contenu de son corps.

Il possède plusieurs attributs :

Attribut	Rôle

var	nom de la variable qui contient l'élément en cours de traitement
items	collection à traiter
varStatus	nom d'un variable qui va contenir des informations sur l'itération en cours de traitement
begin	numéro du premier élément à traiter (le premier possède le numéro 0)
end	numéro du dernier élément à traiter
step	pas des éléments à traiter (par défaut 1)

A chaque itération, la valeur de la variable dont le nom est précisé par la propriété var change pour contenir l'élément de la collection en cours de traitement.

Aucun des attributs n'est obligatoire mais il faut obligatoirement qu'il y ait l'attribut items ou les attributs begin et end.

Le tag forEach peut aussi réaliser des itérations sur les nombres et non sur des éléments d'une collection. Dans ce cas, il ne faut pas utiliser l'attribut items mais uniquement utiliser les attributs begin et end pour fournir les bornes inférieures et supérieures de l'itération.

Exemple :

```
<c:forEach begin="1" end="4" var="i">
<c:out value="${i}"/><br>
</c:forEach>v
```

Résultat :

```
1
2
3
4
```

L'attribut step permet de préciser le pas de l'itération.

Exemple :

```
<c:forEach begin="1" end="12" var="i" step="3">
  <c:out value="${i}"/><br>
</c:forEach>
```

Exemple :

```
1
4
7
10
```

L'attribut varStatus permet de définir une variable qui va contenir des informations sur l'itération en cours d'exécution. Cette variable possède plusieurs propriétés :

Attribut	Rôle
index	indique le numéro de l'occurrence dans l'ensemble de la collection
count	indique le numéro de l'itération en cours (en commençant par 1)
first	booléen qui indique si c'est la première itération
last	booléen qui indique si c'est la dernière itération

Exemple :

```
<c:forEach begin="1" end="12" var="i" step="3" varStatus="vs">
  index = <c:out value="${vs.index}"/> :
  count = <c:out value="${vs.count}"/> :
  value = <c:out value="${i}"/>
  <c:if test="${vs.first}">
     : Premier element
  </c:if>
  <c:if test="${vs.last}">
     : Dernier element
  </c:if>
  <br>
</c:forEach>
```

Résultat :

```
  index = 1 : count = 1 : value = 1 : Premier element
  index = 4 : count = 2 : value = 4
  index = 7 : count = 3 : value = 7
  index = 10 : count = 4 : value = 10 : Dernier element
```

57.3.8. Le tag forTokens

Ce tag permet de découper une chaîne selon un ou plusieurs séparateurs donnés et ainsi d'exécuter de façon répétitive le contenu de son corps autant de fois que d'occurrences trouvées.

Il possède plusieurs attributs :

Attribut	Rôle
var	variable qui contient l'occurrence en cours de traitement (obligatoire)
items	la chaîne de caractères à traiter (obligatoire)
delims	précise le séparateur
varStatus	nom d'un variable qui va contenir des informations sur l'itération en cours de traitement
begin	numéro du premier élément à traiter (le premier possède le numéro 0)
end	numéro du dernier élément à traiter
step	pas des éléments à traiter (par défaut 1)

L'attribut delims peut avoir comme valeur une chaîne de caractères ne contenant qu'un seul caractère (délimiteur unique) ou un ensemble de caractères (délimiteurs multiples).

Exemple :

```
<c:forTokens var="token" items="chaine 1;chaine 2;chaine 3" delims=";">
  <c:out value="${token}" /><br>
</c:forTokens>v
```

Exemple :

```
chaine 1
chaine 2
chaine 3
```

Dans le cas ou il y a plusieurs délimiteurs, chacun peut servir de séparateur

Exemple :

```
<c:forTokens var="token" items="chaine 1;chaine 2,chaine 3" delims=";,">
  <c:out value="${token}" /><br>
</c:forTokens>
```

Attention : Il n'y a pas d'occurrence vide. Dans le cas ou deux séparateurs se suivent consécutivement dans la chaîne à traiter, ceux ci sont considérés comme un seul séparateur. Si la chaîne commence ou se termine par un séparateur, ceux ci sont ignorés.

Exemple :

```
<c:forTokens var="token" items="chaine 1;;chaine 2;;;chaine 3" delims=";">
  <c:out value="${token}" /><br>
</c:forTokens>
```

Résultat :

```
chaine 1
chaine 2
chaine 3
```

Il est possible de ne traiter qu'un sous ensemble des occurrences de la collection. JSTL attribut à chaque occurrence un numéro incrémenter de 1 en 1 à partir de 0. Les attributs begin et end permettent de préciser une plage d'occurrence à traiter.

Exemple :

```
<c:forTokens var="token" items="chaine 1;chaine 2;chaine 3" delims=";" begin="1" end="1" >
  <c:out value="${token}" /><br>
</c:forTokens>
```

Résultat :

```
chaine 2
```

Il est possible de n'utiliser que l'attribut begin ou l'attribut end. Si seul l'attribut begin est précisé alors les n dernières occurrences seront traitées. Si seul l'attribut end est précisé alors seuls les n premières occurrences seront traitées.

Les attributs varStatus et step ont le même rôle que ceux du tag forEach.

57.3.9. Le tag import

Ce tag permet d'accéder à une ressource via son URL pour l'inclure ou l'utiliser dans les traitements de la JSP. La ressource accédée peut être dans une autre application.

Son grand intérêt par rapport au tag <jsp :include> est de ne pas être limité au contexte de l'application web.

Il possède plusieurs attributs :

Attribut	Rôle
url	url de la ressource (obligatoire)
var	nom de la variable qui va stocker le contenu de la ressource sous la forme d'une chaîne de caractère
scope	portée de la variable qui va stocker le contenu de la ressource
context	

	contexte de l'application web qui contient la ressource (si la ressource n'est pas l'application web courante)
charEncoding	jeu de caractères utilisé par la ressource
varReader	nom de la variable qui va stocker le contenu de la ressource sous la forme d'un objet de type java.io.Reader

L'attribut url permet de préciser l'url de la ressource. Cette url peut être relative (par rapport à l'application web) ou absolue.

Exemple :

```
<c:import url="/message.txt" /><br>
```

Par défaut, le contenu de la ressource est inclus dans la JSP. Il est possible de stocker le contenu de la ressource dans une chaîne de caractères en utilisant l'attribut var. Cet attribut attend comme valeur le nom de la variable.

Exemple :

```
<c:import url="/message.txt" var="message" />
<c:out value="${message}" /><BR/>
```

57.3.10. Le tag redirect

Ce tag permet de faire une redirection vers une nouvelle URL.

Les paramètres peuvent être fournis grâce à un ou plusieurs tags fils param.

Exemple :

```
<c:redirect url="liste.jsp">
  <c:param name="id" value="123"/>
</c:redirect>
```

57.3.11. Le tag url

Ce tag permet de formater une url. Il possède plusieurs attributs :

Attribut	Rôle
value	base de l'url (obligatoire)
var	nom de la variable qui va stocker l'url
scope	portée de la variable qui va stocker l'url
context	

Le tag url peut avoir un ou plusieurs tags fils « param ». Le tag param permet de préciser un paramètre et sa valeur pour qu'il soit ajouté à l'url générée.

Le tag param possède deux attributs :

Attribut	Rôle
name	nom du paramètre
value	valeur du paramètre

Exemple :

```
<a href="<c:url url="/index.jsp"/>"/>
```

57.4. La bibliothèque XML

Cette bibliothèque permet de manipuler des données en provenance d'un document XML.

Elle propose les tags suivants répartis dans trois catégories :

Catégorie	Tag
Fondamentale	parse set out
Gestion du flux (condition et itération)	if choose forEach
Transformation XSLT	transform

Les exemples de cette section utilisent un fichier xml nommé personnes.xml dont le contenu est le suivant :

Fichier utilisé dans les exemples :

```
<personnes>
  <personne id="1">
    <nom>nom1</nom>
    <prenom>prenom1</prenom>
  </personne>
  <personne id="2">
    <nom>nom2</nom>
    <prenom>prenom2</prenom>
  </personne>
  <personne id="3">
    <nom>nom3</nom>
    <prenom>prenom3</prenom>
  </personne>
</personnes>
```

L'attribut select des tags de cette bibliothèque utilise la norme Xpath pour sa valeur. JSTL propose une extension supplémentaire à Xpath pour préciser l'objet sur lequel l'expression doit être évaluée. Il suffit de préfixer le nom de la variable par un $

Exemple : recherche de la personne dont l'id est 2 dans un objet nommé listepersonnes qui contient l'arborescence du document xml.

```
$listepersonnes/personnes/personne[@id=2]
```

L'implémentation de JSTL fournie avec le JWSDP utilise Jaxen comme moteur d'interprétation XPath. Donc pour utiliser cette bibliothèque, il faut s'assurer que les fichiers saxpath.jar et jaxen-full.jar soient présents dans le répertoire lib du

répertoire WEB-INF de l'application web.

Pour utiliser cette bibliothèque, il faut la déclarer dans le fichier web.xml du répertoire WEB-INF de l'application web.

Exemple :

```
  <taglib>
    <taglib-uri>http://java.sun.com/jstl/xml</taglib-uri>
    <taglib-location>/WEB-INF/tld/x.tld</taglib-location>
  </taglib>
```

Dans chaque JSP qui utilise un ou plusieurs tags de la bibliothèque, il faut la déclarer avec une directive taglib.

Exemple :

```
<%@ taglib uri="http://java.sun.com/jstl/xml" prefix="x" %>
```

57.4.1. Le tag parse

Le tag parse permet d'analyser un document et de stocker le résultat dans une variable qui pourra être exploité par la JSP ou une autre JSP selon la portée sélectionnée pour le stockage.

Attribut	Rôle
xml	contenu du document à analyser
var	nom de la variable qui va contenir l'arbre DOM générer par l'analyse
scope	portée de la variable qui va contenir l'arbre DOM
varDom	
scopeDom	
filter	
System	

Exemple :

```
<c:import url="/personnes.xml" var="personnes" />
<x:parse xml="${personnes}" var="listepersonnes"  />
```

Dans cet exemple, il suffit simplement que le fichier personnes.xml soit dans le dossier racine de l'application web.

57.4.2. Le tag set

Le tag set est équivalent au tag set de la bibliothèque core. Il permet d'évaluer l'expression Xpath fournie dans l'attribut select et de placer le résultat de cette évaluation dans une variable. L'attribut var permet de préciser la variable qui va recevoir le résultat de l'évaluation sous la forme d'un noeud de l'arbre du document XML.

Il possède plusieurs attributs :

Attribut	Rôle
select	expression XPath à évaluer

var	nom de la variable qui va stocker le résultat de l'évaluation
scope	portée de la variable qui va stocker le résultat

Exemple :

```
<c:import url="/personnes.xml" var="personnes" />
<x:parse xml="${personnes}" var="listepersonnes"  />
<x:set var="unepersonne" select="$listepersonnes/personnes/personne[@id=2]" />
<h1>nom = <x:out select="$unepersonne/nom"/></h1>
```

57.4.3. Le tag out

Le tag out est équivalent au tag out de la bibliothèque core. Il est permet d'évaluer l'expression Xpath fournie dans l'attribut select et d'envoyer le résultat dans le flux de sortie. L'attribut select permet de préciser l'expression Xpath qui doit être évaluée.

Il possède plusieurs attributs :

Attribut	Rôle
select	expression XPath à évaluer
escapeXML	

Exemple : Afficher le nom de la personne dont l'id est 2

```
<c:import url="/personnes.xml" var="personnes" />
<x:parse xml="${personnes}" var="listepersonnes"  />
<x:set var="unepersonne" select="$listepersonnes/personnes/personne[@id=2]" />
<h1><x:out select="$unepersonne/nom"/></h1>
```

Pour stocker le résultat de l'évaluation d'une expression dans une variable, il faut utiliser une combinaison du tag x:out et c:set

Exemple :

```
<c:set var="personneId">
      <x:out select="$listepersonnes/personnes/personne[@id=2]" />
    </c:set>
```

57.4.4. Le tag if

Ce tag est équivalent au tag if de la bibliothèque core sauf qu'il évalue une expression XPath

Il possède plusieurs attributs :

Attribut	Rôle
select	expression XPath à évaluer sous la forme d'un booléen
var	nom de la variable qui va stocker le résultat de l'évaluation
scope	portée de la variable qui va stocker le résultat de l'évaluation

57.4.5. Le tag choose

Ce tag est équivalent au tag choose de la bibliothèque core sauf qu'il évalue des expressions XPath

57.4.6. Le tag forEach

Ce tag est équivalent au tag forEach de la bibliothèque Core. Il permet de parcourir les noeuds issus de l'évaluation d'une expression Xpath.

Il possède plusieurs attributs :

Attribut	Rôle
select	expression XPath à évaluer (obligatoire)
var	nom de la variable qui va contenir le noeud en cours de traitement

Exemple :

```
<c:import url="/personnes.xml" var="personnes" />
<x:parse xml="${personnes}" var="listepersonnes"  />
<x:forEach var="unepersonne" select="$listepersonnes/personnes/*">
    <c:set var="personneId">
      <x:out select="$unepersonne/@id"/>
    </c:set>
    <c:out value="${personneId}" /> - <x:out select="$unepersonne/nom"/>  
       <x:out select="$unepersonne/prenom"/> <br>
</x:forEach>
```

57.4.7. Le tag transform

Ce tag permet d'appliquer une transformation XSLT à un document XML. L'attribut xsl permet de préciser la feuille de style XSL. L'attribut optionnel xml permet de préciser le document xml.

Il possède plusieurs attributs :

Attribut	Rôle
xslt	feuille se style XSLT (obligatoire)
xml	nom de la variable qui contient le document XML à traiter
var	nom de la variable qui va recevoir le résultat de la transformation
scope	portée de la variable qui va recevoir le résultat de la transformation
xmlSystemId	
xsltSystemId	
result	

Exemple :

```
<x:transform xml='${docXml}' xslt='${feuilleXslt}'/>
```

Le document xml à traiter peut être fourni dans le corps du tag

Exemple :

```
<x:transform xslt='${feuilleXslt}'>
  <personnes>
    <personne id="1">
      <nom>nom1</nom>
      <prenom>prenom1</prenom>
    </personne>
    <personne id="2">
      <nom>nom2</nom>
      <prenom>prenom2</prenom>
    </personne>
    <personne id="3">
      <nom>nom3</nom>
      <prenom>prenom3</prenom>
    </personne>
  </personnes>
</x:transform>
```

Le tag transform peut avoir un ou plusieurs noeuds fils param pour fournir des paramètres à la feuille de style XSLT.

57.5. La bibliothèque I18n

Cette bibliothèque facilite l'internationalisation d'une page JSP.

Elle propose les tags suivants répartis dans trois catégories :

Catégorie	Tag
Définition de la langue	setLocale
Formatage de messages	bundle message setBundle
Formatage de dates et nombres	formatNumber parseNumber formatDate parseDate setTimeZone timeZone

Pour utiliser cette bibliothèque, il faut la déclarer dans le fichier web.xml du répertoire WEB-INF de l'application web.

Exemple :

```
<taglib>
    <taglib-uri>http://java.sun.com/jstl/fmt</taglib-uri>
    <taglib-location>/WEB-INF/tld/fmt.tld</taglib-location>
</taglib>
```

Dans chaque JSP qui utilise un ou plusieurs tags de la bibliothèque, il faut la déclarer avec une directive taglib

Exemple :

```
<%@ taglib uri="http://java.sun.com/jstl/fmt" prefix="fmt" %>
```

Le plus simple pour mettre en oeuvre la localisation des messages, c'est de définir un ensemble de fichier qui sont appelé bundle en anglais.

Il faut définir un fichier pour la langue par défaut et un fichier pour chaque langue particulière. Tous ces fichiers ont un préfix commun appelé basename et doivent avoir comme extension .properties. Les fichiers pour les langues particulières doivent le préfix commun suivit d'un underscore puis du code langue et éventuellement d'un underscore suivi du code pays. Ces fichiers doivent être inclus dans le classpath : le plus simple est de les copier dans le répertoire WEB-INF/classes de l'application web.

Exemple :

```
message.properties
message_en.properties
```

Dans chaque fichier, les clés sont identiques, seule la valeur associée à la clé change.

Exemple : le fichier message.properties pour le français (langue par défaut)

```
msg=bonjour
```

Exemple : le fichier message_en.properties pour l'anglais

```
msg=Hello
```

Pour plus d'information, voir le chapitre «L'internationalisation». .

57.5.1. Le tag bundle

Ce tag permet de préciser un bundle à utiliser dans les traitements contenus dans son corps.

Il possède plusieurs attributs :

Attribut	Rôle
baseName	nom de base de ressource à utiliser (obligatoire)
prefix	

Exemple :

```
<fmt:bundle basename="message" >
  <fmt:message key="msg"/>
</fmt:bundle>
```

57.5.2. Le tag setBundle

Ce tag permet de forcer le bundle à utiliser par défaut.

Il possède plusieurs attributs :

Attribut	Rôle
baseName	nom de base de ressource à utiliser (obligatoire)
var	nom de la variable qui va stocker le nouveau bundle
scope	portée de la variable qui va recevoir le nouveau bundle

Exemple :

```
mon message =
<fmt:setBundle basename="message" />
  <fmt:message key="msg"/>
```

57.5.3. Le tag message

Ce tag permet de localiser un message.

Il possède plusieurs attributs :

Attribut	Rôle
key	clé du message à utiliser
bundle	bundle à utiliser
var	nom de la variable qui va recevoir le résultat du formatage
scope	portée de la variable qui va recevoir le résultat du formatage

Pour fournir chaque valeur, il faut utiliser un ou plusieurs tags fils param pour fournir la valeur correspondante.

Exemple :

```
mon message =
<fmt:setBundle basename="message" />
  <fmt:message key="msg"/>
```

Résultat :

```
mon message = bonjour
```

Si aucune valeur n'est trouvée pour la clé fournie alors le tag renvoie ???XXX ??? où XXX représente le nom de la clé.

Exemple :

```
mon message =
<fmt:setBundle basename="message" />
  <fmt:message key="test"/>
```

Résultat :

```
mon message = ???test???
```

57.5.4. Le tag setLocale

Ce tag permet de sélectionner une nouvelle Locale.

Exemple :

```
<fmt:setLocale value="en"/>
mon message =
<fmt:setBundle basename="message" />
  <fmt:message key="msg"/>
```

Résultat :

```
mon message = Hello
```

57.5.5. Le tag formatNumber

Ce tag permet de formater des nombres selon la locale. L'attribut value permet de préciser la valeur à formater. L'attribut type permet de préciser le type de formatage à réaliser.

Il possède plusieurs attributs :

Attribut	Rôle
value	valeur à formater
type	CURRENCY ou NUMBER ou PERCENT
pattern	format personnalisé
currencyCode	code de la monnaie à utiliser pour le type CURRENCY
currencySymbol	symbole de la monnaie à utiliser pour le type CURRENCY
groupingUsed	booléen pour préciser si les nombres doivent être groupés
maxIntegerDigits	nombre maximum de chiffre dans la partie entière
minIntegerDigits	nombre minimum de chiffre dans la partie entière
maxFractionDigits	nombre maximum de chiffre dans la partie décimale
minFractionDigits	nombre minimum de chiffre dans la partie décimale
var	nom de la variable qui va stocker le résultat
scope	portée de la variable qui va stocker le résultat

Exemple :

```
<c:set var="montant" value="12345.67" />
montant = <fmt:formatNumber value="${montant}" type="currency"/>
```

57.5.6. Le tag parseNumber

Ce tag permet de convertir une chaîne de caractère qui contient un nombre en une variable décimale.

Il possède plusieurs attributs :

Attribut	Rôle
value	valeur à traiter
type	CURRENCY ou NUMBER ou PERCENT
parseLocale	Locale à utiliser lors du traitement
integerOnly	booléen qui indique si le résultat doit être un entier (true) ou un flottant (false)
pattern	format personnalisé
var	nom de la variable qui va stocker le résultat

scope	portée de la variable qui va stocker le résultat

Exemple : convertir en entier un identifiant passé en paramètre de la requête

```
<fmt:parseNumber value="${param.id}" var="id"/>
```

57.5.7. Le tag formatDate

Ce tag permet de formater des dates selon la locale.

Il possède plusieurs attributs :

Attribut	Rôle
value	valeur à formater
type	DATE ou TIME ou BOTH
dateStyle	FULL ou LONG ou MEDIUM ou SHORT ou DEFAULT
timeStyle	FULL ou LONG ou MEDIUM ou SHORT ou DEFAULT
pattern	format personnalisé
timeZone	timeZone utilisé pour le formatage
var	nom de la variable qui va stocker le résultat
scope	portée de la variable qui va stocker le résultat

L'attribut value permet de préciser la valeur à formater. L'attribut type permet de préciser le type de formatage à réaliser. L'attribut dateStyle permet de préciser le style du formatage.

Exemple :

```
<jsp:useBean id="now" class="java.util.Date" />
Nous sommes le <fmt:formatDate value="${now}" type="date" dateStyle="full"/>.
```

57.5.8. Le tag parseDate

Ce tag permet d'analyser une chaîne de caractères contenant une date pour créer un objet de type java.util.Date.

Il possède plusieurs attributs :

Attribut	Rôle
value	valeur à traiter
type	DATE ou TIME ou BOTH
dateStyle	FULL ou LONG ou MEDIUM ou SHORT ou DEFAULT
timeStyle	FULL ou LONG ou MEDIUM ou SHORT ou DEFAULT
pattern	format personnalisé
parseLocale	Locale utilisé pour le formatage
timeZone	timeZone utilisé pour le formatage

var	nom de la variable de type java.util.date qui va stocker le résultat
scope	portée de la variable qui va stocker le résultat

57.5.9. Le tag setTimeZone

Ce tag permet de stocker un fuseau horaire dans une variable.

Il possède plusieurs attributs :

Attribut	Rôle
value	fuseau horaire à stocker (obligatoire)
var	nom de la variable de stockage
scope	portée de la variable de stockage

57.5.10. Le tag timeZone

Ce tag permet de préciser un fuseau horaire particulier à utiliser dans son corps.

Il possède plusieurs attributs :

Attribut	Rôle
value	chaine de caractère ou objet java.util.TimeZone qui précise le fuseau horaire à utiliser

57.6. La bibliothèque Database

Cette bibliothèque facilite l'accès aux bases de données. Son but n'est pas de remplacer les accès réalisés grâce à des beans ou des EJB mais de fournir une solution simple mais non robuste pour accéder à des bases de données. Ceci est cependant particulièrement utile pour développer des pages de tests ou des prototypes.

Elle propose les tags suivants répartis dans deux catégories :

Catégorie	Tag
Définition de la source de données	setDataSource
Exécution de requêtes SQL	query transaction update

Pour utiliser cette bibliothèque, il faut la déclarer dans le fichier web.xml du répertoire WEB-INF de l'application web.

Exemple :

```
  <taglib>
    <taglib-uri>http://java.sun.com/jstl/sql</taglib-uri>
    <taglib-location>/WEB-INF/tld/sql.tld</taglib-location>
</taglib>
```

Dans chaque JSP qui utilise un ou plusieurs tags de la bibliothèque, il faut la déclarer avec une directive taglib

Exemple :

```
<%@ taglib uri="http://java.sun.com/jstl/sql" prefix="sql" %>
```

57.6.1. Le tag setDataSource

Ce tag permet de créer une connexion vers la base de données à partir des données fournies dans les différents attributs du tag.

Il possède plusieurs attributs :

Attribut	Rôle
driver	nom de la classe du pilote JDBC à utiliser
source	url de la base de données à utiliser
user	nom de l'utilisateur à utiliser lors de la connexion
password	mot de passe de l'utilisateur à utiliser lors de la connexion
var	nom de la variable qui va stocker l'objet créé lors de la connexion
scope	portée de la variable qui va stocker l'objet créé
dataSource	

Exemple : accéder à une base via ODBC dont le DNS est test

```
<sql:setDataSource driver="sun.jdbc.odbc.JdbcOdbcDriver" url="jdbc:odbc:test"
user="" password=""/>
```

57.6.2. Le tag query

Ce tag permet de réaliser des requêtes de sélection sur une source de données.

Il possède plusieurs attributs :

Attribut	Rôle
sql	requête SQL à exécuter
var	nom de la variable qui stocke les résultats de l'exécution de la requête
scope	portée de la variable qui stocke le résultat
startRow	numéro de l'occurrence de départ à traiter
maxRow	nombre maximum d'occurrence à stocker
dataSource	connexion particulière à la base de données à utiliser

L'attribut sql permet de préciser la requête à exécuter :

Exemple :

```
<sql:query var="reqPersonnes" sql="SELECT * FROM personnes" />
```

Le résultat de l'exécution de la requête est stocké dans un objet qui implémente l'interface `javax.servlet.jsp.jstl.sql.Result dont le nom est donné via l'attribut var`

L'interface Result possède cinq getter :

Méthode	Rôle
String[] getColumnNames()	renvoie un tableau de chaines de caractères qui contient le nom des colonnes
int getRowCount()	renvoie le nombre d'enregistrements trouvé lors de l'exécution de la requête
Map[] getRows()	renvoie une collection qui associe à chaque colonne la valeur associé pour l'occurrence en cours
Object[][] getRowsByIndex()	renvoie un tableau contenant les colonnes et leur valeur
boolean isLimitedByMaxRows()	renvoie un booléen qui indique si le résultat de la requête a été limité

Exemple : connaitre le nombre d'occurrence renvoyées par la requête

```
<p>Nombre d'enregistrement trouvé : <c:out value="${reqPersonnes.rowCount}" /></p>
```

La requête SQL peut être précisée avec l'attribut sql ou dans le corps du tag

Exemple :

```
<sql:query var="reqPersonnes" >
  SELECT * FROM personnes
</sql:query>
```

Le tag forEach de la bibliothèque core est particulièrement utile pour itérer sur chaque occurrence retournée par la requête SQL.

Exemple :

```
<TABLE border="1" CELLPadding="4" cellspacing="0">
<TR>
<td>id</td>
<td>nom</td>
<td>prenom</td>
</TR>

<c:forEach var="row" items="${reqPersonnes.rows}" >
<TR>
<td><c:out value="${row.id}" /></td>
<td><c:out value="${row.nom}" /></td>
<td><c:out value="${row.prenom}" /></td>
</TR>
</c:forEach>
</TABLE>
```

Il est possible de fournir des valeurs à la requête SQL. Il faut remplacer dans la requête SQL la valeur par le caractère ?. Pour fournir, la ou les valeurs il faut utiliser un ou plusieurs tags fils param.

Le tag param possède un seul attribut :

Attribut	Rôle

value	valeur de l'occurrence correspondante dans la requête SQL

Pour les valeurs de type date, il faut utiliser le tag dateParam.

Le tag dateParam possède plusieurs attributs :

Attribut	Rôle
value	objet de type java.util.date qui contient la valeur de la date (obligatoire)
type	format de la date : TIMESTAMP ou DATE ou TIME

Exemple :

```
<c:set var="id" value="2" />

<sql:query var="reqPersonnes" >
  SELECT * FROM personnes where id = ?
            <sql:param value="${id}" />
</sql:query>
```

57.6.3. Le tag transaction

Ce tag permet d'encapsuler plusieurs requêtes SQL dans une transaction.

Il possède plusieurs attributs :

Attribut	Rôle
dataSource	connexion particulière à la base de données à utiliser
isolation	READCOMMITTED ou READUNCOMMITTED ou REPEATABLEREAD ou SERIALIZABLE

57.6.4. Le tag update

Ce tag permet de réaliser une mise à jour grâce à une requête SQL sur la source de données.

Il possède plusieurs attributs :

Attribut	Rôle
sql	requête SQL à exécuter
var	nom de la variable qui stocke le nombre d'occurrence impactée par l'exécution de la requête
scope	portée de la variable qui stocke le nombre d'occurrence impactée
dataSource	connexion particulière à la base de données à utiliser

Exemple :

```
<c:set var="id" value="2" />
<c:set var="nouveauNom" value="nom 2 modifié" />

<sql:update var="nbRec">
UPDATE personnes
SET nom = ?
```

```
WHERE id=?
<sql:param value="${nouveauNom}"/>
<sql:param value="${id}"/>
</sql:update>

<p>nb enregistrement modifiés = <c:out value="${nbRec}"/></p>
```

58. Struts

Struts est un framework pour applications web développé par le projet Jakarta de la fondation Apache. C'est le plus populaire des frameworks pour le développement d'applications web avec Java.

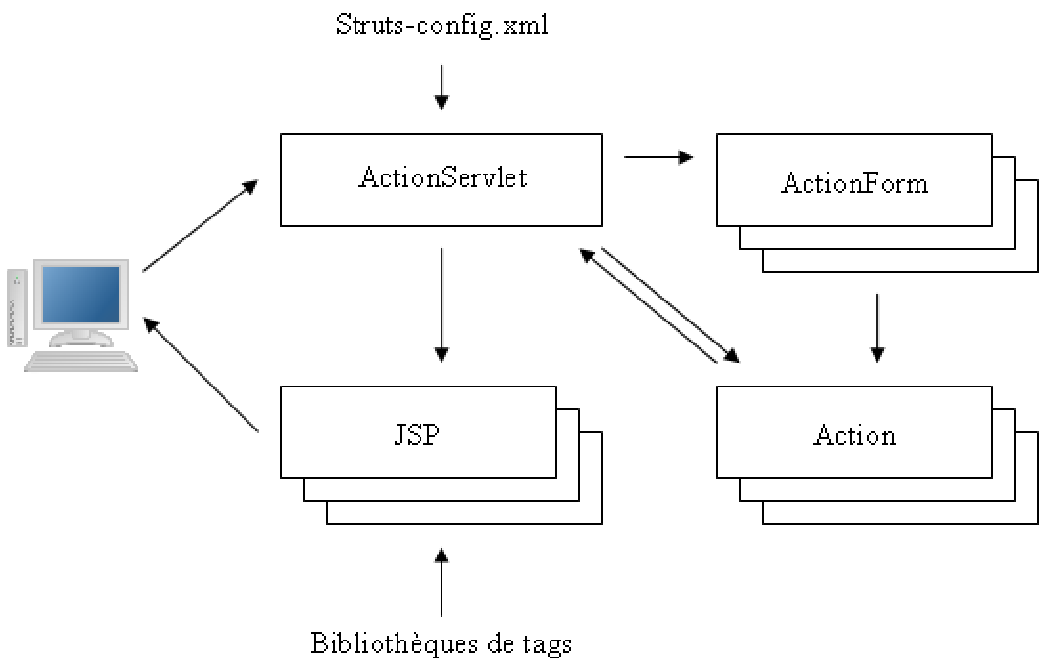
Il a été initialement développé par Craig Mc Clanahan qui l'a donné au projet Jakarta d'Apache en mai 2000. Depuis, Struts a connu un succès grandissant auprès de la communauté du libre et des développeurs à tel point qu'il sert de base à de nombreux autres framework open source et commerciaux et que la plupart des grands IDE propriétaires (Borland, IBM, BEA, ...) intègrent une partie dédiée à son utilisation.

Struts met en oeuvre le modèle MVC 2 basé sur une seule servlet faisant office de contrôleur et des JSP pour l'IHM. L'application de ce modèle permet une séparation en trois parties distinctes de l'interface, des traitements et des données de l'application.

Struts se concentre sur la vue et le contrôleur. L'implémentation du modèle est laissée libre aux développeurs : ils ont le choix d'utiliser des java beans, un outil de mapping objet/relationnel, des EJB ou toute autre solution.

Pour le contrôleur, Struts propose une unique servlet par application qui lit la configuration de l'application dans un fichier au format XML. Cette servlet de type ActionServlet reçoit toutes les requêtes de l'utilisateur concernant l'application. En fonction du paramétrage, elle instancie un objet de type Action qui contient les traitements et renvoie une valeur particulière à la servlet. Celle ci permet de déterminer la JSP qui affichera le résultat des traitements à l'utilisateur.

Les données issues de la requête sont encapsulées dans un objet de type ActionForm. Struts va utiliser l'introspection pour initialiser les champs de cet objet à partir des valeurs fournies dans la requête.



Struts utilise un fichier de configuration au format XML (struts-config.xml) pour connaître le détail des éléments qu'il va gérer dans l'application et comment ils vont interagir lors des traitements.

Pour la vue, Struts utilise par défaut des JSP avec un ensemble de plusieurs bibliothèques de tags personnalisés pour faciliter leur développement.

Struts propose aussi plusieurs services techniques : pool de connexion aux sources de données, internationalisation, ...

La dernière version ainsi que toutes les informations utiles peuvent être obtenues sur le site http://jakarta.apache.org/struts/

Il existe plusieurs versions de Struts : 1.0 (publiée en juin 2001), 1.1 et 1.2

Ce chapitre contient plusieurs sections :

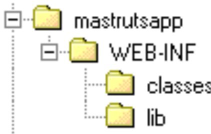
- L'installation et la mise en oeuvre
- Le développement des vues
- La configuration de Struts
- Les bibliothèques de tags personnalisés
- La validation de données

58.1. L'installation et la mise en oeuvre

Il faut télécharger la dernière version de Struts sur le site du projet Jakarta. La version utilisée dans cette section est la version 1.2.4.

Il suffit de décompresser le fichier jakarta-struts-1.2.4.zip dans un répertoire quelconque du système d'exploitation.

Il faut créer une structure de répertoire qui va accueillir l'application web, nommée par exemple mastrutsapp :



En utilisant Tomcat, une mise en oeuvre possible est de créer le répertoire de base de l'application dans le répertoire webapps.

Pour pouvoir utiliser Struts dans une application web, il faut copier les fichiers *.jar contenus dans le répertoire lib de Struts dans le répertoire WEB-INF/lib de l'application :

- commons-beanutils.jar
- commons-collection.jar
- commons-digester.jar
- commons-fileupload
- commons-logging.jar
- commons-validator.jar
- jakarta-oro.jar
- struts.jar

Il faut aussi copier les fichiers .tld (struts-bean.tld, struts-html.tld, struts-logic.tld, struts-nested.tld, struts-tiles.tld) dans le répertoire WEB-INF ou un de ses sous-répertoires.

Dans le répertoire WEB-INF, il faut créer deux fichiers :

- web.xml : le descripteur de déploiement de l'application
- struts-config.xml : le fichier de configuration de Struts

Le fichier web.xml minimal est le suivant :

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   version="2.4">
  <display-name>Mon application Struts de tests</display-name>

  <!-- Servlet controleur de Struts -->
  <servlet>
    <servlet-name>action</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
    <init-param>
      <param-name>config</param-name>
      <param-value>/WEB-INF/struts-config.xml</param-value>
    </init-param>
    <init-param>
      <param-name>debug</param-name>
      <param-value>2</param-value>
    </init-param>
    <init-param>
      <param-name>detail</param-name>
      <param-value>2</param-value>
    </init-param>
    <load-on-startup>2</load-on-startup>
  </servlet>

  <!-- Mapping des url avec la servlet -->
  <servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>*.do</url-pattern>
  </servlet-mapping>

  <!-- page d'accueil de l'application -->
  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>

  <jsp-config>
    <!-- Descripteur des bibliotheques personnalisees de Struts -->
    <taglib>
      <taglib-uri>/struts-bean</taglib-uri>
      <taglib-location>/WEB-INF/struts-bean.tld</taglib-location>
    </taglib>

    <taglib>
      <taglib-uri>/struts-html</taglib-uri>
      <taglib-location>/WEB-INF/struts-html.tld</taglib-location>
    </taglib>

    <taglib>
      <taglib-uri>/struts-logic</taglib-uri>
      <taglib-location>/WEB-INF/struts-logic.tld</taglib-location>
    </taglib>

    <taglib>
      <taglib-uri>/struts-nested</taglib-uri>
      <taglib-location>/WEB-INF/struts-nested.tld</taglib-location>
    </taglib>

    <taglib>
      <taglib-uri>/struts-tiles</taglib-uri>
      <taglib-location>/WEB-INF/struts-tiles.tld</taglib-location>
    </taglib>
  </jsp-config>

</web-app>
```

Le mapping des url de l'application prend généralement une des deux formes suivantes :

- préfixer chaque url

* suffixer chaque url avec une extension

Exemple de prefixe d'url :

```
<servlet-mapping>
  <servlet-name>action</servlet-name>
  <url-pattern>/do/*</url-pattern>
</servlet-mapping>
```

Exemple de suffixe d'url :

```
<servlet-mapping>
  <servlet-name>action</servlet-name>
  <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

Les exemples fournis sont de simples exemples : n'importe quel préfixe ou extension peut être utilisé avec leur forme respective.

Le fichier struts-config.xml minimal est le suivant :

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts-config PUBLIC
   "-//Apache Software Foundation//DTD Struts Configuration 1.2//EN"
   "http://jakarta.apache.org/struts/dtds/struts-config_1_2.dtd">
<struts-config>

</struts-config>
```

Ces deux fichiers seront complétés au fur à mesure des sections suivantes.

Comme Struts met en oeuvre le modèle MVC, il est possible de développer séparément les différents composants de l'application.

58.1.1. Un exemple très simple

L'exemple de cette section va simplement demander le nom et le mot de passe de l'utilisateur et le saluer si ces deux données saisies ont une valeur précise.

Cet exemple est particulièrement simple et sera enrichi dans les autres sections de ce chapitre : son but est de proposer un exemple simple d'enchainement de deux pages et de récupération des données d'un formulaire.

Le fichier struts-config.xml va contenir la définition des entités utilisées dans l'exemple : le Form Bean et l'Action.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts-config
PUBLIC "-//Apache Software Foundation//DTD Struts Configuration 1.0//EN"
"http://jakarta.apache.org/struts/dtds/struts-config_1_2.dtd">
<struts-config>

  <form-beans type="org.apache.struts.action.ActionFormBean">
    <form-bean name="loginForm" type="com.jmd.test.struts.data.LoginForm" />
  </form-beans>

  <action-mappings type="org.apache.struts.action.ActionMapping">
    <action path="/login" parameter="" input="/index.jsp" scope="request"
          name="loginForm" type="com.jmd.test.struts.controleur.LoginAction">
      <forward name="succes" path="/accueil.jsp" redirect="false" />
```

```
      <forward name="echec" path="/index.jsp" redirect="false" />
    </action>
  </action-mappings>

</struts-config>
```

Il faut écrire la page d'authentification.

Exemple : la page index.jsp

```
<%@ page language="java" %>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>
<html:html locale="true">
  <head>
    <title>Authentification</title>
    <html:base/>
  </head>
  <body bgcolor="white">
    <html:form action="login" focus="nomUtilisateur">
      <table border="0" align="center">
        <tr>
          <td align="right">
            Utilisateur :
          </td>
          <td align="left">
            <html:text property="nomUtilisateur" size="20" maxlength="20"/>
          </td>
        </tr>
        <tr>
          <td align="right">
            Mot de Passe :
          </td>
          <td align="left">
            <html:password property="mdpUtilisateur" size="20" maxlength="20"
                          redisplay="false"/>
          </td>
        </tr>
        <tr>
          <td align="right">
            <html:submit property="submit" value="Submit"/>
          </td>
          <td align="left">
            <html:reset/>
          </td>
        </tr>
      </table>
    </html:form>
  </body>
</html:html>
```

Il faut aussi définir la page d'accueil qui sera affichée une fois l'utilisateur authentifié.

Exemple : la page accueil.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html:html locale="true">
  <head>
    <title>Accueil</title>
    <html:base/>
    <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
  </head>
  <body bgcolor="white">
```

```
    <h1> Bienvenue <bean:write name="loginForm" property="nomUtilisateur"/></h1>
  </body>
</html:html>
```

Il faut définir l'objet de type ActionForm qui va encapsuler les données saisies par l'utilisateur dans la page d'authentification.

Exemple : la classe LoginForm

```
package com.jmd.test.struts.data;

import org.apache.struts.action.*;
import javax.servlet.http.HttpServletRequest;

public class LoginForm extends ActionForm {
  String nomUtilisateur;

  String mdpUtilisateur;

  public String getMdpUtilisateur() {
    return mdpUtilisateur;
  }

  public void setMdpUtilisateur(String mdpUtilisateur) {
    this.mdpUtilisateur = mdpUtilisateur;
  }

  public String getNomUtilisateur() {
    return nomUtilisateur;
  }

  public void setNomUtilisateur(String nomUtilisateur) {
    this.nomUtilisateur = nomUtilisateur;
  }

  public ActionErrors validate(ActionMapping mapping, HttpServletRequest request) {
    ActionErrors errors = new ActionErrors();
    return errors;
  }

  public void reset(ActionMapping mapping, HttpServletRequest request) {
    this.mdpUtilisateur = null;
    this.nomUtilisateur = null;
  }

}
```

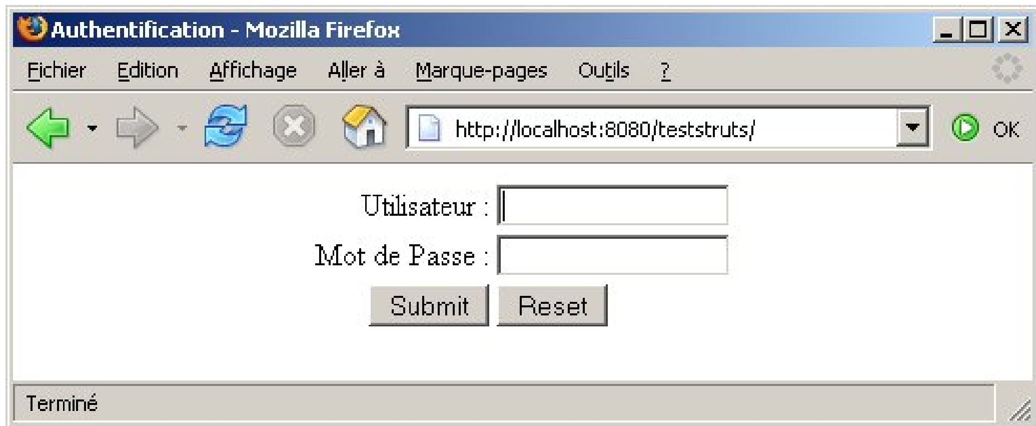
Enfin, il faut définir un objet de type Action qui va encapsuler les traitements lors de la soumission du formulaire.

Exemple : la classe LoginAction

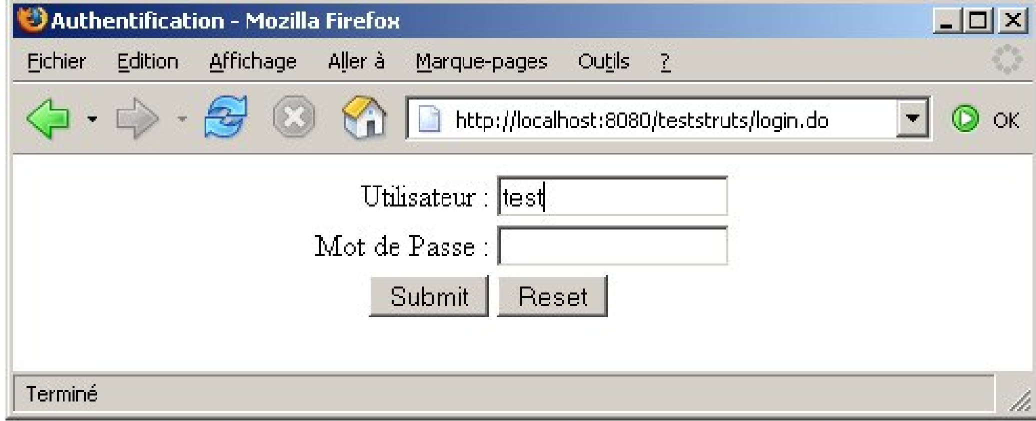
```
package com.jmd.test.struts.controleur;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;

import com.jmd.test.struts.data.LoginForm;

public final class LoginAction extends Action {

  public ActionForward execute(ActionMapping mapping,
                               ActionForm form,
                               HttpServletRequest req,
                               HttpServletResponse res) throws Exception {
```

```
    String resultat = null;
    String nomUtilisateur = ((LoginForm) form).getNomUtilisateur();
    String mdpUtilisateur = ((LoginForm) form).getMdpUtilisateur();

    if (nomUtilisateur.equals("xyz") && mdpUtilisateur.equals("xyz")) {
      resultat = "succes";
    } else {
      resultat = "echec";
    }

    return mapping.findForward(resultat);
  }

}
```

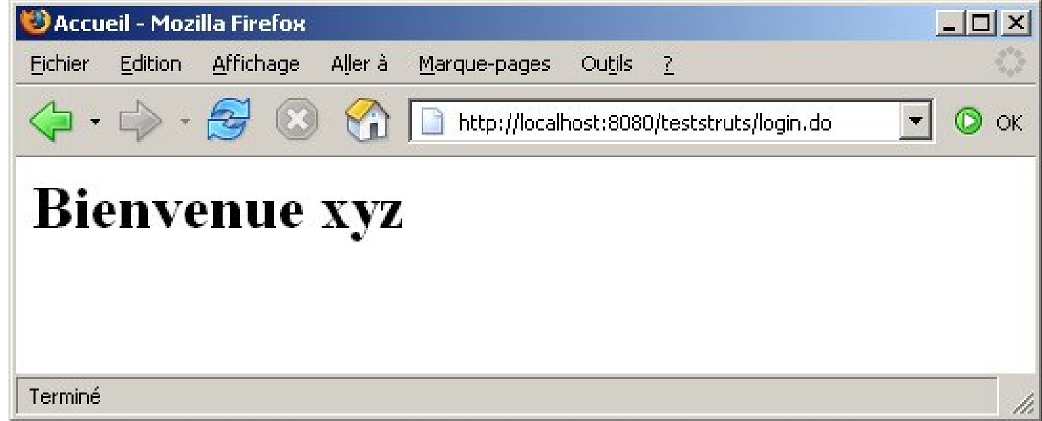
Pour exécuter cet exemple, il faut le déployer dans un conteneur web (par exemple Tomcat)



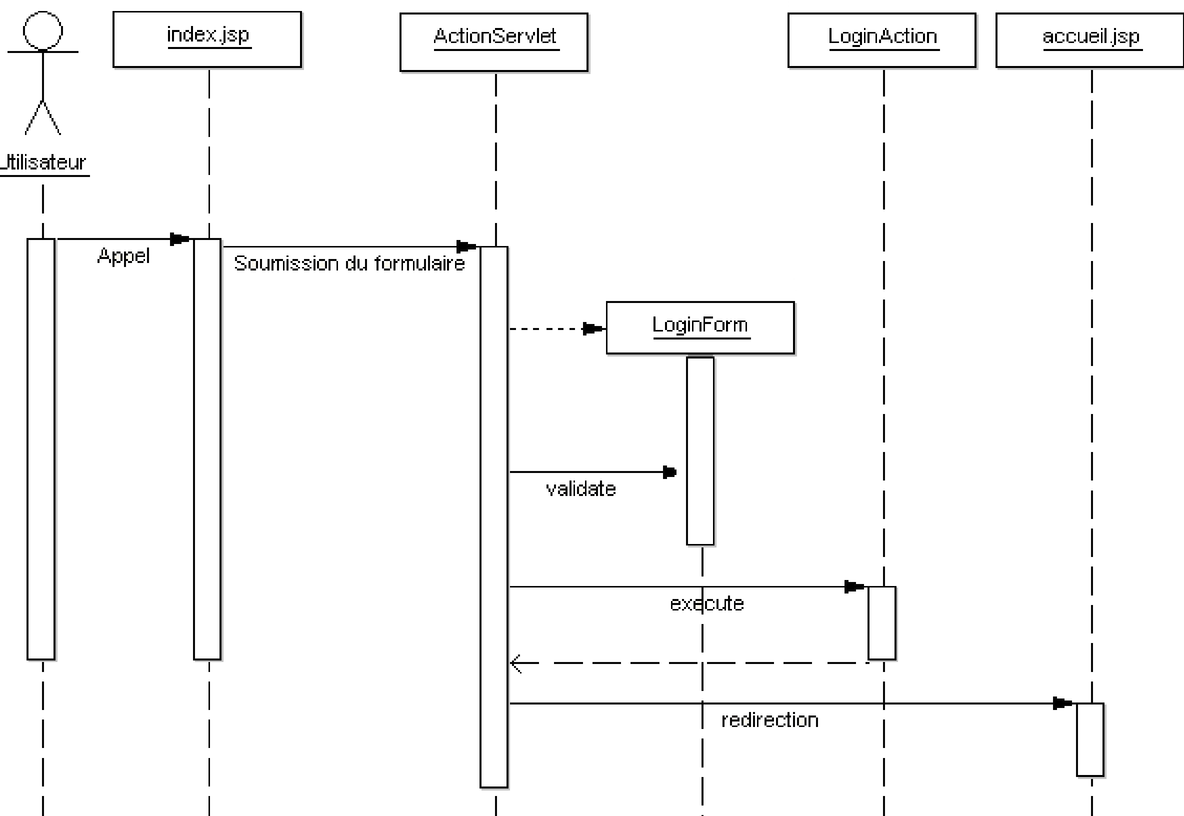
Si le nom d'utilisateur et le mot de passe saisis ne valent pas « xyz » alors la page d'authentification est réaffichée.



Si le nom d'utilisateur et le mot de passe saisie valent « xyz » alors la page d'accueil s'affiche.



Le diagramme de séquence ci-dessous résume les principales actions de cet exemple.



L'utilisateur appelle la page d'authentification index.jsp, saisie son nom d'utilisateur et son mot de passe et valide le formulaire.

L'ActionServlet intercepte la requête pour la traiter en effectuant les actions suivantes :

- Instancie un objet de type LoginForm et alimente ses données avec celles correspondantes dans la requête
- Appel de la méthode validate de la classe LoginForm pour valider les données saisies par l'utilisateur
- Détermination de l'Action à utiliser en fonction des informations contenues dans le fichier struts-config.xml. Dans l'exemple, c'est un objet de type LoginAction.
- Appel de la méthode execute() de la classe LoginAction qui contient les traitements à effectuer pour répondre à la requête. Elle renvoie un objet de type ActionForward
- L'ActionServlet détermine la page à afficher à l'utilisateur en réponse en fonction de la valeur renvoyée par la méthode execute() et des informations du fichier de configuration
- La page déterminée est retournée au navigateur de l'utilisateur pour être affichée

58.2. Le développement des vues

Les vues représentent l'interface entre l'application et l'utilisateur. Avec le framework Struts, les vues d'une application web sont constituées par défaut de JSP et de pages HTML.

Pour faciliter leur développement, Struts propose un ensemble de nombreux tags personnalisés regroupés dans plusieurs bibliothèques de tags personnalisés possédant chacun un thème particulier :

- HTML : permet de faciliter le développement de page Web en HTML
- Bean : permet de faciliter l'utilisation des Javabeans
- Logic : permet de faciliter la mise en oeuvre de la logique des traitements d'affichage
- Tiles : permet la gestion de modèles (templates)

Struts propose aussi au travers de ces tags de nombreuses fonctionnalités pour faciliter le développement : un formatage des données, une gestion des erreurs, ...

58.2.1. Les objets de type ActionForm

Un objet de type ActionForm est un objet respectant les spécifications des JavaBeans qui permet à Struts de mapper automatiquement les données saisies dans une page HTML avec les attributs correspondants dans l'objet. Il peut aussi réaliser une validation des données saisies par l'utilisateur.

Pour automatiser cette tâche, Struts utilise l'introspection pour rechercher un accesseur correspondant au nom du paramètre contenant la donnée dans la requête HTTP.

C'est la servlet faisant office de contrôleur qui instancie un objet de type ActionForm et alimente ses propriétés avec les valeurs contenues dans la requête émise à partir de la page.

Pour chaque page contenant des données à utiliser, il faut définir un objet qui hérite de la classe abstraite org.apache.struts.action.ActionForm. Par convention, le nom de cette classe est le nom de la page suivi de "Form".

Pour chaque donnée, il faut définir un attribut private ou protected qui contiendra la valeur, un getter et un setter public en respectant les normes de développement des Java beans.

Exemple :

```
package com.jmd.test.struts.data;

import org.apache.struts.action.*;
import javax.servlet.http.HttpServletRequest;

public class LoginForm extends ActionForm {
  String nomUtilisateur;

  String mdpUtilisateur;

  public String getMdpUtilisateur() {
    return mdpUtilisateur;
  }

  public void setMdpUtilisateur(String mdpUtilisateur) {
    this.mdpUtilisateur = mdpUtilisateur;
  }

  public String getNomUtilisateur() {
    return nomUtilisateur;
  }

  public void setNomUtilisateur(String nomUtilisateur) {
    this.nomUtilisateur = nomUtilisateur;
  }

...

}
```

La méthode reset() doit être redéfinie pour initialiser chaque attribut avec une valeur par défaut. Cette méthode est appelée par l'ActionServlet lorsqu'une instance de l'ActionForm est obtenue par la servlet et avant que cette dernière ne valorise les propriétés.

Exemple :

```
  public void reset(ActionMapping mapping, HttpServletRequest request) {
    this.mdpUtilisateur = null;
    this.nomUtilisateur = null;
  }
```

La signature de cette méthode est la suivante :
public void reset(ActionMapping mapping, HttpServletRequest request);

La méthode validate() peut être redéfinie pour permettre de réaliser des traitements de validation des données contenues dans l'ActionForm

La signature de cette méthode est la suivante :
public ActionErrors validate(ActionMapping mapping, HttpServletRequest request);

Elle renvoie une instance de la classe ActionErrors qui encapsule les différentes erreurs détectées ou renvoie null si aucune erreur n'est rencontrée.

Exemple :

```
public ActionErrors validate(ActionMapping mapping, HttpServletRequest request) {
  ActionErrors errors = new ActionErrors();

  if ((nomUtilisateur == null) || (nomUtilisateur.length() == 0))
    errors.add("nomUtilisateur", new ActionError("erreur.nomutilisateur.obligatoire"));

  if ((mdpUtilisateur == null) || (mdpUtilisateur.length() == 0))
    errors.add("mdpUtilisateur", new ActionError("erreur.mdputilisateur.obligatoire"));

  return errors;
}
```

Comme les objets de type ActionForm sont des éléments de la vue du modèle MVC, les objets de type ActionForm ne doivent contenir aucun traitement métier. La méthode validate() ne doit contenir que des contrôles de surface (présence de données, taille des données, format des données, ...).

Il faut compiler cette classe et la placer dans le répertoire WEB-INF/classes suivi de l'arborescence correspondant au package de la classe.

Il faut aussi déclarer pour chaque ActionForm, un tag <form-bean> dans le fichier struts-config.xml. Ce tag possède plusieurs attributs :

Attribut	Rôle
Name	le nom sous lequel Struts va connaître l'objet
Type	le type complètement qualifié de la classe de type ActionForm

Exemple :

```
<form-beans type="org.apache.struts.action.ActionFormBean">
  <form-bean name="loginForm" type="com.jmd.test.struts.data.LoginForm" />
</form-beans>
```

Chaque objet de type ActionForm doit être défini dans un tag <form-beans> et <form-bean> dans le fichier de description struts-config.xml.

Pour demander l'exécution des traitements de validation des données, il est nécessaire d'utiliser l'attribut validate dans le fichier struts-config.xml.

Remarque : pour assurer un découplage entre la partie IHM et la partie métier, il n'est pas recommandé de passer à cette dernière une instance de type ActionForm. Il est préférable d'utiliser un objet dédié respectant le modèle de conception Data Transfert Object (DTO).

58.2.2. Les objets de type DynaActionForm

Le développement d'objet de type ActionForm pour chaque page peut s'avérer fastidieux à écrire (même si des outils peuvent se charger de générer les getters et les setters nécessaires) et surtout à maintenir dans le cas d'une évolution. Ceci est d'autant plus vrai si cet objet n'est utilisé que pour obtenir les données du formulaire.

Struts propose les objets de type DynaActionForm qui permettent selon une déclaration dans le fichier de configuration d'obtenir dynamiquement les données sans avoir à développer explicitement un objet dédié.

Les DynaActionForm doivent donc obligatoirement être déclarés dans le fichier de configuration struts-config.xml comme les ActionForm.

Exemple :

```
<form-beans>
  <form-bean name="saisirProduitActionForm"
             type="org.apache.struts.action.DynaActionForm">
    <form-property name="reference" type="java.lang.String"/>
    <form-property name="libelle" type="java.lang.String"/>
    <form-property name="prix" type="java.lang.String" initial="0"/>
  </form-bean>
</form-beans>
```

Par défaut la méthode validate() de la classe DynaActionForm ne réalise aucun traitement. Pour pouvoir l'utiliser, il est nécessaire de créer une classe fille qui va hériter de DynaActionForm dans laquelle la méthode validate() va être redéfinie. C'est cette classe fille qui devra alors être précisée dans l'attribut type du tag <form-bean>.

58.3. La configuration de Struts

L'essentiel de la configuration de Struts se fait dans le fichier de configuration struts-config.xml.

58.3.1. Le fichier struts-config.xml

Ce fichier au format XML contient le paramétrage nécessaire à l'exécution d'une application utilisant Struts.

Il doit se nommer struts-config.xml et il doit être dans le répertoire WEB-INF de l'application.

Le tag racine de ce document XML est le tag <struts-config>.

Ce fichier se compose de plusieurs parties :

- la déclaration des beans de formulaire (ActionForm) dans un tag <form-beans>
- la déclaration des redirections globales à toute l'application dans un tag <global-forwards>
- la déclaration des Action dans un tag <action-mappings>
- la déclaration des ressources dans un ou plusieurs tags <message-ressources>
- la déclaration des plug-ins dans un ou plusieurs tags <plug-in>

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts-config
PUBLIC "-//Apache Software Foundation//DTD Struts Configuration 1.0//EN"
"http://jakarta.apache.org/struts/dtds/struts-config_1_2.dtd">
<struts-config>

  <form-beans type="org.apache.struts.action.ActionFormBean">
    <form-bean name="loginForm" type="com.jmd.test.struts.data.LoginForm" />
  </form-beans>

  <action-mappings type="org.apache.struts.action.ActionMapping">
    <action path="/login" parameter="" input="/index.jsp" scope="request"
          name="loginForm" type="com.jmd.test.struts.controleur.LoginAction">
      <forward name="succes" path="/accueil.jsp" redirect="false" />
      <forward name="echec" path="/index.jsp" redirect="false" />
```

```
    </action>
  </action-mappings>

</struts-config>
```

Le tag <form-beans> permet de définir les objets de type ActionForm et DynaActionForm utilisée dans l'application.

Les DynaActionForm sont déclarés grâce à un tag <form-bean> fils du tag <form-beans>. Comme pour les ActionForm, le paramètre name permet de préciser le nom qui va faire référence au bean. L'attribut type doit avoir comme valeur org.apache.struts.action.DynaActionForm ou une classe pleinement qualifiée qui en hérite.

Chaque attribut du bean doit être déclaré dans un tag fils <form-property>. Ce tag possède plusieurs attributs :

- name : nom de la propriété
- type : type pleinement qualifié de la propriété suivi de [] pour un tableau
- size : taille si le type est un tableau
- initial : permet de préciser la valeur initiale de la propriété

Exemple :

```
  <form-beans>
    <form-bean name="saisirProduitActionForm"
               type="org.apache.struts.action.DynaActionForm">
      <form-property name="reference" type="java.lang.String"/>
      <form-property name="libelle" type="java.lang.String"/>
      <form-property name="prix" type="java.lang.String" initial="0"/>
    </form-bean>
  </form-beans>
```

Le tag <global-exception> permet de définir des handlers globaux à l'application pour traiter des exceptions.

Le tag <action-mappings> permet de définir l'ensemble des actions de l'application. Celles-ci sont unitairement définies grâce à un tag <action>.

Le tag Action permet d'associer une URL (/login.do dans l'exemple) avec un objet de type Action (LoginAction dans l'exemple). Ainsi, à chaque utilisation de cette URL, l'ActionServlet utilise la classe Action associée pour exécuter les traitements.

La propriété path permet d'indiquer l'URI d'appel de ce mapping : c'est cette valeur qui sera par exemple indiquée (suffixée ou préfixée selon le paramétrage du fichier web.xml) dans l'attribut action d'un formulaire ou href d'un lien.

La propriété type permet d'indiquer le nom pleinement qualifié de la classe Action qui sera utilisée par ce mapping.

La propriété name permet d'indiquer le nom d'un bean de type ActionForm associé à ce mapping. Cet objet encapsulera les données contenues dans la requête http.

La propriété scope permet de préciser la portée de l'objet ActionForm instancié par l'ActionServlet précisé par l'attribut name :

- request : la durée de vie des données ne concerne que la requête
- session : les données concernent un utilisateur
- application : les données sont communes à tous les utilisateurs de l'application

Il est préférable d'utiliser la portée la plus courte possible et d'éviter l'utilisation de la portée application.

L'attribut validate permet de préciser si les données de l'ActionForm doivent être validées en faisant appel à la méthode validate(). La valeur par défaut est true.

La propriété input permet de préciser l'URI de la page de saisie des données qui sera réaffichée en cas d'échec de la validation des données.

Le tag fils <forward> permet de préciser avec l'attribut path l'URI d'une page qui sera affichée lorsque l'Action renverra la valeur précisée dans l'attribut name. L'attribut redirect permet de préciser le type de redirection qui sera effectuée (redirect si la valeur est true sinon c'est un forward qui sera effectué). L'URI fournie doit être relative dans le cas d'un forward et relative ou absolue dans le cas d'un redirect.
Les informations contenues dans ce tag seront utilisés lors de l'instanciation d'objets de type ActionForward

Le tag <global-forward> permet de définir des redirections communes à toute l'application. Ce tag utilise des tags fils de type <forward>. Les redirections définies localement sont prioritaires par rapport à celles définies de façon globales.

Le tag <message-ressources> permet de définir les ressources nécessaires à l'internationalisation de l'application.

Le tag <plug-in> permet de configurer des plug-ins de Struts tel que Tiles ou Validator.

Le tag <data-sources> permet de définir des sources de données. Chaque source de données est définie dans un tag <data-source>.

58.3.2. La classe ActionMapping

La classe ActionMapping encapsule les données définies dans un tag <Action> du fichier de configuration.

Chacune de ces ressources est définie dans le fichier de configuration struts-config.xml dans un tag <action> regroupé dans un tag <action-mappings>.

La méthode findForward() permet d'obtenir une redirection définie dans un tag <forward> de l'action ou dans un tag <global-forward>.

La classe ActionMappings encapsule une collection d'objets de type ActionMapping.

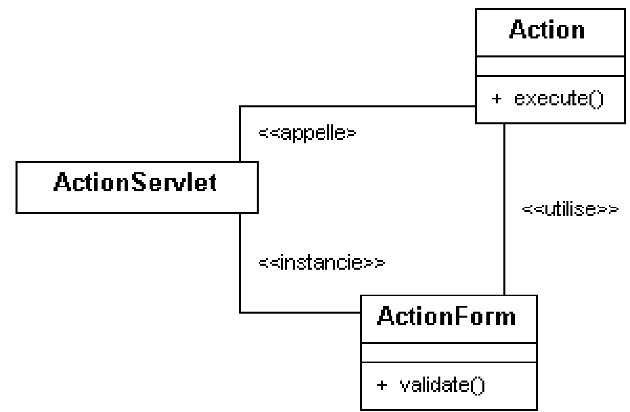
58.3.3. Le développement de la partie contrôleur

Basée sur le modèle MVC 2, la partie contrôleur de Struts se compose donc de deux éléments principaux dans une application Struts :

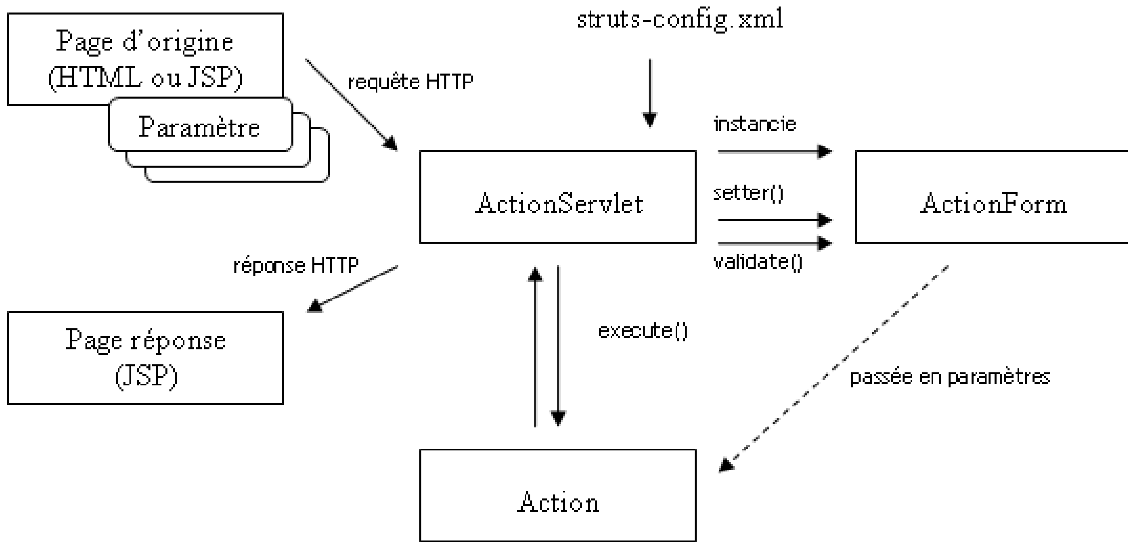
- une servlet de type org.apache.struts.action.ActionServlet
- plusieurs classes de type org.apache.struts.action.Action

La partie contrôleur est implémentée en utilisant une seule et unique servlet par application. Cette servlet doit hériter de la classe org.apache.struts.action.ActionServlet.

Cette servlet possède des traitements génériques qui utilisent les informations contenues dans le fichier struts-config.xml et dans des objets du type org.apache.struts.action.Action



Une instance de la classe RequestProcessor est utilisée par l'ActionServlet en appelant sa méthode process() pour initialiser un objet de type ActionForm associé à l'action lié à la requête en cours de traitement.



L'ActionServlet vérifie la présence d'une instance du type de l'ActionForm dans la session : dans la négative, une nouvelle instance est créée et ajoutée à la session. La clé associée au bean dans la session est définie par l'attribut attribute du tag <Action>.

La requête est ensuite analysée : pour chaque attribut présent dans la requête, la servlet recherche dans l'ActionForm une propriété dont le nom correspond en utilisant l'introspection : si elle est trouvée, la servlet appelle son setter pour lui associer la valeur contenue dans la requête. La correspondance des noms doit être exacte en respectant la casse.

Si la validation est positionnée dans le fichier de configuration, la servlet appelle la méthode validate() de l'ActionForm. Si la validation réussie et ou n'est pas demandée, l'ActionForm est passé en paramètre de la méthode execute() de l'instance d'Action.

58.3.4. La servlet de type ActionServlet

Le coeur d'une application Struts est composé d'une servlet de type org.apache.struts.action.ActionServlet.

Cette servlet reçoit les requêtes HTTP émises par le client et en fonction de celles ci, elle appelle un objet du type Action qui lui est associé dans le fichier struts-config.xml. Le traitement d'une requête par une application Struts suit plusieurs étapes :

1. le navigateur client envoie une requête
2. réception de la requête par la servlet de type ActionServlet
3. en fonction de l'URI et du fichier de configuration struts-config.xml, la servlet instancie ou utilise l'objet de type ActionForm précisé. La servlet utilise l'introspection pour appeler les setters des propriétés dont le nom des propriétés correspond
4. la servlet instancie un objet de type Action associé à l'URI de la requête
5. la servlet appelle la méthode execute() de la classe Action. En retour de cet appel un objet de type ActionMapping permet d'indiquer à la servlet la page JSP qui sera affichée en réponse
6. la JSP génère la réponse HTML qui sera affichée sur le navigateur client

Pour respecter les spécifications J2EE, cette servlet doit être définie dans le fichier de déploiement web.xml de l'application web.

58.3.5. La classe Action

Un objet de type Action contient une partie spécifique de la logique métier de l'application : il est chargé de traiter ces données et déterminer quelle sera la page à afficher en fonction des traitements effectués.

Cet objet doit étendre la classe org.apache.struts.action.Action. Par convention, le nom de cette classe est le nom de la page suivi de "Action".

Il est important de développer ces classes de façon thread-safe : le contrôleur utilise une même instance pour traiter simultanément plusieurs requêtes. Il n'est donc pas recommandé d'utiliser des variables d'instances pour stocker des données sur une requête.

La méthode la plus importante de cette classe est la méthode execute(). C'est elle qui doit contenir les traitements qui seront exécutés. Depuis la version 1.1 de Struts, elle remplace la méthode perform() qui est deprecated mais toujours présente pour des raisons de compatibilité. La différence majeure entre la méthode perform() et execute() est que cette dernière déclare la possibilité de lever une exception.

La méthode execute() attend plusieurs paramètres :

- Un objet de type ActionMapping
- Un objet de type ActionForm
- Un objet de type HttpServletRequest
- Un objet de type HtppServletResponse

Il existe une autre surcharge de la méthode execute() qui attend les mêmes paramètres sauf pour les deux derniers paramètres qui sont de type ServletRequest et ServletResponse.

Les traitements typiquement réalisés dans cette méthode sont les suivants :

- Utiliser un cast vers l'objet de type ActionForm à utiliser pour l'objet fourni en paramètre de la méthode : ceci permet un accès aux données spécifiques de l'objet de type ActionForm
- Réaliser les traitements requis sur ces données
- Déterminer la page de retour en fonction des traitements réalisés sous la forme d'un objet de type ActionForward.

Une bonne pratique de développement consiste à faire réaliser les traitements par des objets métiers dédiés indépendant de l'API Struts. Ces objets peuvent par exemple être des Javabeans ou des EJB.

Pour obtenir un objet de type ActionForward encapsulant la page réponse, il faut utiliser la méthode findForward() de l'objet de type ActionMapping passé en paramètre de la méthode execute(). La méthode findForward() attend en paramètre le nom de la page tel qu'il est défini dans le fichier struts-config.xml.

Cet objet est retourné au contrôleur qui assurera la redirection vers la page concernée.

Pour stocker les éventuelles erreurs rencontrées, il est nécessaire de créer une instance de la classe ActionErrors

Exemple :

```
ActionErrors erreurs = new ActionErrors();
```

Pour extraire les données issues de l'objet ActionForm, il est nécessaire d'effectuer un cast vers le type de l'instance fournie en paramètre

Exemple :

```
    String nomUtilisateur = "";
    String mdpUtilisateur = "";

    if (form != null) {
      nomUtilisateur = ((LoginForm) form).getNomUtilisateur();
      mdpUtilisateur = ((LoginForm) form).getMdpUtilisateur();
    }
```

Pour extraire les données issues d'un objet de type DynaActionForm, il est nécessaire d'effectuer un cast vers le type DynaActionForm de l'instance fournie en paramètre.

Comme les objets de type DynaActionForm ne possèdent pas de getter et setter, pour obtenir la valeur d'une propriété d'un tel objet il est nécessaire d'utiliser la méthode get() en passant en paramètre le nom de la propriété et de caster la valeur retournée.

Exemple :

```
DynaActionForm daf = (DynaActionForm)form;

String reference = (String)daf.get("reference");
String libelle = (String)daf.get("libelle");
int prix = Integer.parseInt( (String)daf.get("prix")  );
```

Si une erreur est détectée dans les traitements, il faut instancier un objet de type ActionError et la fournir en paramètre avec le type de l'erreur à la méthode add() de l'instance de type ActionErrors.

Exemple :

```
if (nomUtilisateur.equals("xyz") && mdpUtilisateur.equals("xyz")) {
  resultat = "succes";
} else {
  erreurs.add(ActionErrors.GLOBAL_ERROR, new ActionError("erreur.login.invalide"));
  resultat = "echec";
}
```

A la fin des traitements de la méthode execute(), si des erreurs ont été ajoutées il est nécessaire de faire appel à la méthode saveErrors() pour enregistrer les erreurs.

Exemple :

```
if (!erreurs.isEmpty()) {
  saveErrors(req, erreurs);
}
```

Pour permettre un affichage des erreurs, il faut faire renvoyer à la méthode une instance de la classe ActionForward() qui encapsule la page émettrice de la requête.

Exemple :

```
return (new ActionForward(mapping.getInput()));
```

Sans erreur, le dernier traitement à réaliser est la création d'une instance de type ActionForward qui désignera la page à afficher en réponse à la requête.

Il y a deux façons d'obtenir cette instance :

- instancier directement un objet de type ActionForward
- utiliser la méthode findForward() de l'instance de type ActionMapping fournie en paramètre de la méthode execute()

Il existe plusieurs constructeurs pour la classe ActionForward dont les deux principaux sont :

- ActionForward(String path)
- ActionForward(String path, boolean redirect)

Le paramètre direct est un booléen qui avec la valeur true fera procéder à une redirection vers la réponse (Response.sendRedirect()) et qui avec la valeur false fera procéder à un transfert vers la page réponse (RequestDispatcher.forward()).

L'utilisation de l'instance de type ActionMapping est sûrement la façon la plus pratique. Un appel à la méthode findForward() en précisant en paramètre le nom logique défini dans le fichier struts-config.xml permet d'obtenir un objet de type ActionForward pointant vers la page associée au nom logique.

Exemple :

```
    return mapping.findForward(resultat);
```

A partir de l'objet de type HttpRequest, il est possible d'accéder à la session en utilisant la méthode getSession().

Exemple :

```
HttpSession session = request.getSession();
session.setAttribute(" key ", user);
```

58.3.6. La classe DispatchAction

La classe DispatchAction permet d'associer plusieurs actions à un même formulaire. Cette situation est assez fréquente par exemple lorsqu'une page propose l'ajout, la modification et suppression de données.

Elle va permettre en une seule action de réaliser une des opérations supportées par l'action. L'opération à réaliser selon l'action qui est sélectionnée par l'utilisateur doit être fournie dans la requête http sous la forme d'un champ caché de type Hidden ou en paramètre dans l'url.

Exemple :

```
<%@ page contentType="text/html;charset=windows-1252"%>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean"%>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html"%>
<html:html locale="true">
  <html>
    <head>
      <meta http-equiv="Content-Type" content="text/html; charset=windows-1252"/>
      <title>untitled</title>
      <SCRIPT language="javascript" type="text/javascript">
      function setOperation(valeur){
        document.forms[0].operation.value=valeur;
      }
    </SCRIPT>
    </head>
    <body>
      <html:form action="operations.do" focusIndex="reference">
        <html:hidden property="operation" value="aucune"/>
        <table>
          <tr>
            <td>
              <bean:message key="app.saisirproduit.libelle.reference"/>:
            </td>
            <td>
              <html:text property="reference"/>
            </td>
          </tr>
          <tr>
            <td colspan="2" align="center">
              <html:submit onclick="setOperation('ajouter');">Ajouter</html:submit>
              <html:submit onclick="setOperation('modifier');">Modifier</html:submit>
              <html:submit onclick="setOperation('supprimer');">Supprimer</html:submit>
            </td>
          </tr>
```

```
        </table>
      </html:form>
    </body>
  </html>
</html:html>
```

L'implémentation de l'action doit hériter de la classe DispatchAction. Il est inutile de redéfinir la méthode execute() mais il faut définir autant de méthodes nommées avec les valeurs possibles des opérations.

L'introspection sera utilisée pour déterminer dynamiquement la méthode à appeler en fonction de l'opération reçue dans la requête.

Exemple :

```
package test.struts.controleur;

import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.actions.DispatchAction;

import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class OperationsAction extends DispatchAction
{

  public ActionForward ajouter(
    ActionMapping       mapping,
    ActionForm          form,
    HttpServletRequest  request,
    HttpServletResponse response) throws IOException, ServletException
  {
    System.out.println("Appel de la methode ajouter()");
    return (mapping.findForward("succes"));
  }

  public ActionForward modifier(
    ActionMapping       mapping,
    ActionForm          form,
    HttpServletRequest  request,
    HttpServletResponse response) throws IOException, ServletException
  {
    System.out.println("Appel de la methode modifier()");
    return (mapping.findForward("succes"));
  }

  public ActionForward supprimer(
    ActionMapping       mapping,
    ActionForm          form,
    HttpServletRequest  request,
    HttpServletResponse response) throws IOException, ServletException
  {
    System.out.println("Appel de la methode supprimer()");
    return (mapping.findForward("succes"));
  }
}
```

Dans le fichier de configuration strut-config.xml, il faut déclarer l'action en précisant dans un attribut parameter le nom du paramètre de la requête qui contient l'opération à réaliser.

Exemple :

```
<struts-config>
...
```

```
  <form-beans>
...
    <form-bean name="operationsForm"
               type="org.apache.struts.action.DynaActionForm">
      <form-property name="operation" type="java.lang.String"/>
      <form-property name="reference" type="java.lang.String"/>
    </form-bean>
...
  </form-beans>
  <action-mappings>
...
    <action path="/operations" type="test.struts.controleur.OperationsAction"
            name="operationsForm" scope="request" validate="true" parameter="operation">
      <forward name="succes" path="/operations.jsp"/>
    </action>
...
  </action-mappings>
...
</struts-config>
```

Si la méthode à invoquer n'est pas définie dans la classe de type DispatchAction, alors une exception est levée

Exemple :

```
03-juil.-2006 13:14:43 org.apache.struts.actions.DispatchAction dispatchMethod
GRAVE: Action[/operations] does not contain method named supprimer
java.lang.NoSuchMethodException: test.struts.controleur.OperationsAction.supprimer(
org.apache.struts.action.ActionMapping, org.apache.struts.action.ActionForm,
javax.servlet.http.HttpServletRequest, javax.servlet.http.HttpServletResponse)
        at java.lang.Class.getMethod(Class.java)
```

Il est aussi possible d'utiliser plusieurs boutons avec pour valeur l'opération à réaliser. Ceci évite d'avoir à écrire du code Javascript. Dans ce cas, chaque bouton doit avoir comme valeur de l'attribut property la valeur fournie à l'attribut parameter du tag <action>.

Exemple :

```
<%@ page contentType="text/html;charset=windows-1252"%>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean"%>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html"%>
<html:html locale="true">
  <html>
    <head>
      <meta http-equiv="Content-Type" content="text/html; charset=windows-1252"/>
      <title>untitled</title>
    </head>
    <body>
      <html:form action="operations.do" focusIndex="reference">
        <table>
          <tr>
            <td>
              <bean:message key="app.saisirproduit.libelle.reference"/>:
            </td>
            <td>
              <html:text property="reference"/>
            </td>
          </tr>
          <tr>
            <td colspan="2" align="center">
              <html:submit property="operation">ajouter</html:submit>
              <html:submit property="operation">modifier</html:submit>
              <html:submit property="operation">supprimer</html:submit>
            </td>
          </tr>
        </table>
      </html:form>
    </body>
  </html>
```

```
</html:html>
```

Attention cependant, la valeur du bouton est aussi son libellé : il est donc nécessaire de synchroniser le nom du bouton dans la vue et la méthode correspondante dans l'action. Ceci empêche l'internationalisation du libellé du bouton.

58.3.7. La classe LookupDispatchAction

Pour contourner le problème de l'internationalisation des opérations avec DispatchAction sans Javascript, il est possible d'utiliser une action de type LookupDispatchAction.

Dans ce cas, le mapping ne se fait pas sur une valeur en dur mais sur la valeur d'une clé extraite des RessourcesBundles en fonction de la Locale courante.

La déclaration dans le fichier de configuration est similaire à celle nécessaire pour l'utilisation d'une action de type DispatchAction.

Exemple :

```
...
<struts-config>
  <form-beans>
..
    <form-bean name="operationsLookupForm"
               type="org.apache.struts.action.DynaActionForm">
      <form-property name="operation" type="java.lang.String"/>
      <form-property name="reference" type="java.lang.String"/>
    </form-bean>
...
  </form-beans>
  <action-mappings>
...
    <action path="/operationslookup" type="test.struts.controleur.OperationsLookupAction"
            name="operationsLookupForm" scope="request" validate="true" parameter="operation">
      <forward name="succes" path="/operationslookup.jsp"/>
    </action>
...
  </action-mappings>
...
</struts-config>
```

Dans la vue, le libellé des boutons de chaque action doit être défini dans les RessourcesBundles.

Exemple :

```
<%@ page contentType="text/html;charset=windows-1252"%>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean"%>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html"%>
<html:html locale="true">
  <html>
    <head>
      <meta http-equiv="Content-Type" content="text/html; charset=windows-1252"/>
      <title>Test LookupDispatchAction</title>
    </head>
    <body>
      <html:form action="operationslookup.do" focusIndex="reference">
        <table>
          <tr>
            <td>
              <bean:message key="app.saisirproduit.libelle.reference"/>:
            </td>
            <td>
              <html:text property="reference"/>
            </td>
          </tr>
```

```
          <tr>
            <td colspan="2" align="center">
              <html:submit property="operation">
                <bean:message key="operation.ajouter"/>
              </html:submit>
              <html:submit property="operation">
                <bean:message key="operation.modifier"/>
              </html:submit>
              <html:submit property="operation">
                <bean:message key="operation.supprimer"/>
              </html:submit>
            </td>
          </tr>
        </table>
      </html:form>
    </body>
  </html>
</html:html>
```

La valeur de chaque bouton doit être identique et précisée dans l'attribut property.

Il faut définir dans les ResourceBundles les libellés des boutons de chaque opération.

Exemple : ApplicationResources.properties

```
...
operation.ajouter   = Ajouter
operation.modifier  = Modifier
operation.supprimer = Supprimer
...
```

Exemple : ApplicationResources_en.properties

```
...
operation.ajouter   = Add
operation.modifier  = Modify
operation.supprimer = Delete
...
```

L'action doit hériter de la classe LookupDispatchAction. Il faut redéfinir la méthode getKeyMethodMap() pour qu'elle renvoie une collection de type Map dont la valeur de chaque clé correspond à la clé du ResourceBundle du bouton et la valeur correspond à la méthode correspondante à invoquer.

La définition des méthodes de chaque opération est identique à celle utilisée avec une action de type DispatchAction.

Exemple :

```
package test.struts.controleur;

import java.util.HashMap;
import java.util.Map;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;

import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.struts.actions.LookupDispatchAction;
import org.apache.struts.util.MessageResources;

public class OperationsLookupAction extends LookupDispatchAction
{
  public static final String OPERATION_AJOUTER = "operation.ajouter";
  public static final String OPERATION_MODIFIER = "operation.modifier";
```

```
  public static final String OPERATION_SUPPRIMER = "operation.supprimer";


 public Map getKeyMethodMap() {
  Map map = new HashMap();
  map.put(OPERATION_AJOUTER, "ajouter");
  map.put(OPERATION_MODIFIER, "modifier");
  map.put(OPERATION_SUPPRIMER, "supprimer");
  return map;
}

  public ActionForward ajouter(
    ActionMapping       mapping,
    ActionForm          form,
    HttpServletRequest  request,
    HttpServletResponse response) throws IOException, ServletException
  {
    System.out.println("Appel de la methode ajouter()");
    return (mapping.findForward("succes"));
  }

  public ActionForward modifier(
    ActionMapping       mapping,
    ActionForm          form,
    HttpServletRequest  request,
    HttpServletResponse response) throws IOException, ServletException
  {
    System.out.println("Appel de la methode modifier()");
    return (mapping.findForward("succes"));
  }

  public ActionForward supprimer(
    ActionMapping       mapping,
    ActionForm          form,
    HttpServletRequest  request,
    HttpServletResponse response) throws IOException, ServletException
  {
    System.out.println("Appel de la methode supprimer()");
    return (mapping.findForward("succes"));
  }
}
```

Grâce à la méthode getKeyMethodMap(), la valeur de chaque opération est déterminée dynamiquement en fonction de la Locale.

58.3.8. La classe ForwardAction

Cette action permet uniquement une redirection vers une page sans qu'aucun traitement ne soit exécuté.

L'intérêt est de centraliser ces redirections dans le fichier de configuration plutôt que de les laisser en dur dans la ou les pages qui en ont besoin.

Il suffit de définir une action dans le fichier struts-config.xml en utilisant les attributs :

- path : l'URI de l'action
- type : org.apache.struts.actions.ForwardAction
- parameter : la page vers laquelle l'utilisateur va être redirigé

Exemple :

```
    <action path="/redirection"
            type="org.apache.struts.actions.ForwardAction"
            parameter="/test.jsp">
```

```
    </action>
```

Pour utiliser cette action, il suffit de faire un lien vers le path de l'action

Exemple :

```
<html:link action="redirection.do">Page de test</html:link>
```

58.4. Les bibliothèques de tags personnalisés

L'utilisation des bibliothèques de tags de Struts nécessite la définition des bibliothèques dans le fichier de déploiement web.xml et la déclaration des bibliothèques utilisées dans chaque page.

Exemple :

```
  <%@ taglib uri="/WEB-INF/struts-html.tld"  prefix="html" %>
  <%@ taglib uri="/WEB-INF/struts-bean.tld"  prefix="bean" %>
  <%@ taglib uri="/WEB-INF/struts-logic.tld"  prefix="logic" %>
```

Les vues sont aussi composées selon le modèle MVC d'objets de type ActionForm ou DynaActionForm qui encapsulent les données d'une page. Ils permettent l'échange de données entre la vue et les objets métiers via le contrôleur.

58.4.1. La bibliothèque de tags HTML

Cette bibliothèque permet de faciliter le développement de page Web en HTML.

Pour utiliser cette bibliothèque, il faut, comme pour toute bibliothèque de tags personnalisés, réaliser plusieurs opérations :

1. copier le fichier struts-html.tld dans le répertoire WEB-INF de la webapp
2. configurer le fichier WEB-INF/web.xml pour déclarer la bibliothèque de tag

```
<taglib>
  <taglib-uri>struts-html.tld</taglib-uri>
  <taglib-location>/WEB-INF/struts-html.tld</taglib-location>
</taglib>
```

3. ajouter dans chaque page JSP qui va utiliser cette bibliothèque un tag de directive taglib précisant l'utilisation de la bibliothèque

```
<%@ taglib uri="struts-html.tld" prefix="html" %>
```

La plupart de ces tags encapsulent des tags HTML notamment pour les formulaires mais ils assurent aussi des traitements particuliers à Struts.

Exemple :

Un lien vers une url absolu avec HTML doit intégrer le nom de la webapp :
<a href="/testwebapp/index.jsp">

La balise Struts correspondante sera indépendante de la webapp : elle tient compte automatiquement du contexte de l'application
<html:link page="/index.jsp">

Il est cependant préférable d'utiliser un mapping défini dans le fichier struts-config.xml plutôt que d'utiliser un lien vers la page JSP correspondante. Ceci va permettre l'exécution de l'Action correspondante.

Exemple :

<html:link page="/index.do">Accueil</html:link>

Tag	*Description*
base	Encapsule un tag HTML <base>
button	Encapsule un tag HTML <input type="button">
cancel	Encapsule un tag HTML <input type="submit"> avec la valeur Cancel
checkbox	Encapsule un tag HTML <input type="checkbox">
errors	Affiche les messages d'erreurs stockés dans la session
file	Encapsule un tag HTML <input type="file">
form	Encapsule un tag HTML <form>
frame	Encapsule un tag HTML <frame>
hidden	Encapsule un tag HTML <input type="hidden">
html	Encapsule un tag HTML <html>
image	Encapsule une action affichée sous la forme d'une image
img	Encapsule un tag HTML <img>
javascript	Assure la génération du code Javascript requis par le plug-in Validator
link	Encapsule un tag HTML <A>
messages	Affiche les messages stockés dans la session
multibox	Assure le rendu de plusieurs checkbox
option	encapsule un tag HTML <option>
options	Assure le rendu de plusieurs options
optionsCollection	Assure le rendu de plusieurs options
password	Encapsule un tag HTML <input type="password">
radio	Encapsule un tag HTML <input type="radio">
reset	Encapsule un tag HTML <input type="reset">
rewrite	Le rendu d'une URI
select	Encapsule un tag HTML <select>
submit	Encapsule un tag HTML <input type="submit">
text	Encapsule un tag HTML <input type="text">
textarea	Encapsule un tag HTML <input type="textarea">
xhtml	Le rendu des tags HTML est au format XHTML

Les tags les plus utilisés seront détaillés dans les sections suivantes.

58.4.1.1. Le tag <html:html>

Ce tag génère un tag HTML <html>.

Il possède plusieurs attributs dont les principaux sont :

Attribut	Rôle
lang	génère un attribut lang en fonction de celle stockée dans la session, ou dans la requête ou de la Locale par défaut en fonction de leur présence respective
locale	utilise la valeur true pour forcer le stockage dans la session de la Locale correspondant à la langue de la requête Ce tag est deprecated depuis la version 1.2 car il créé automatique une session : utiliser l'attribut lang à la place
xhtml	utilise la valeur true pour assurer un rendu au format xhtml des tags

Ce tag doit être inclus dans un tag <html:form>

58.4.1.2. Le tag <html:form>

Ce tag génère un tag HTML <form>.

Il possède de nombreux attributs correspondant aux attributs du tag html <form> dont les principaux sont :

Attribut	Rôle
action	url vers laquelle le formulaire sera soumis
enctype	type d'encodage du formulaire lors de la soumission
focus	nom de l'élément qui aura le focus au premier affichage de la page
method	méthode de soumission du formulaire
name	nom associé à la classe ActionForm
scope	portée de la classe ActionForm
target	cible d'affichage de la réponse
type	type de la classe ActionForm

58.4.1.3. Le tag <html:button>

Ce tag génère un tag HTML <input> de type button.

Il possède de nombreux attributs dont les principaux sont :

Attribut	Rôle
alt	correspond à l'attribut alt du tag HTML
altKey	clé du ResourceBundle dont la valeur sera affectée à l'attribut alt du tag HTML
bundle	nom du bean qui encapsule le ResourceBundle (utilisé lorsque que plusieurs ResourceBundles sont définis)
disabled	true pour rendre le bouton non opérationnel
property	nom du paramètre dans la requête http lors de la soumission du formulaire : correspond à l'attribut name du tag HTML

title	correspond à l'attribut title du tag HTML
titleKey	clé du ResourceBundle dont la valeur sera affectée à l'attribut title du tag HTML
value	libellé du bouton : correspond à l'attribut value du tag HTML

Ce tag doit être inclus dans un tag <html:form>

Exemple :

```
<html:button property="valider" value="Valider" title="Valider les données" />
```

Résultat

```
<input type="button" name="valider" value="Valider" title="Valider les données">
```

58.4.1.4. Le tag <html:cancel>

Ce tag génère un tag HTML <input> de type button avec une valeur spécifique pour permettre d'identifier ce bouton comme étant celui de type "Cancel".

Il possède des attributs similaires au tag <html:button>.

Il n'est pas recommandé d'utiliser l'attribut property : il faut laisser la valeur par défaut de Struts pour lui permettre d'identifier ce bouton. La valeur par défaut de l'attribut property permet à Struts de déterminer la valeur de retour de la méthode Action.isCancelled.

Exemple :

```
<html:cancel />
```

Résultat

```
<input type="submit" name="org.apache.struts.taglib.html.CANCEL" value="Cancel" onclick="bCancel=true;">
```

58.4.1.5. Le tag <html:submit>

Ce tag génère un tag HTML <input type="submit"> permettant la validation d'un formulaire.

Il possède des attributs similaires au tag <html:button>.

Ce tag doit être inclus dans un tag <html:form>

Exemple :

```
<html:submit />
```

Résultat

```
<input type="submit" value="Submit">
```

58.4.1.6. Le tag <html:radio>

Ce tag génère un tag HTML <input type="radio"> permettant d'afficher un bouton radio.

Exemple :

```
<html:radio property="sexe" value="femme" />Femme<br>
<html:radio property="sexe" value="homme" />Homme<br>
```

Résultat :

```
<input type="radio" name="sexe" value="femme">Femme<br>
<input type="radio" name="sexe" value="homme">Homme<br>
```

58.4.1.7. Le tag <html:checkbox>

Ce tag génère un tag HTML <input type="checkbox"> permettant d'afficher un bouton de type case à cocher.

Exemple :

```
<html:checkbox property="caseACocher"> Une case a cocher</html:checkbox>
```

Résultat :

```
<input type="checkbox" name="caseACocher" value="on">Une case a cocher
```

58.4.2. La bibliothèque de tags Bean

Cette bibliothèque fournit des tags pour faciliter la gestion et l'utilisation des javabeans.

Pour utiliser cette bibliothèque, il faut, comme pour toute bibliothèque de tags personnalisés, réaliser plusieurs opérations :

1. copier le fichier struts-bean.tld dans le répertoire WEB-INF de la webapp
2. configurer le fichier WEB-INF/web.xml pour déclarer la bibliothèque de tag

```
<taglib>
  <taglib-uri>struts-bean.tld</taglib-uri>
  <taglib-location>/WEB-INF/struts-bean.tld</taglib-location>
</taglib>
```

3. ajouter dans chaque page JSP qui va utiliser cette bibliothèque un tag de directive taglib précisant l'utilisation de la bibliothèque

```
<%@ taglib uri="struts-bean.tld" prefix="bean" %>
```

58.4.2.1. Le tag <bean:cookie>

Le tag <bean:cookie> permet d'obtenir la ou les valeurs d'un cookie.

Il possède plusieurs attributs :

Attribut	Rôle
id	identifiant du cookie
name	nom du cookie
multiple	précise si toutes les valeurs ou seulement la première valeur du cookie sont retournées
value	valeur du cookie si celui n'existe pas : dans ce cas, il est créé

58.4.2.2. Le tag <bean:define>

Le tag <bean:define> permet de définir une variable.

Il possède plusieurs attributs :

Attribut	Rôle
id	nom de la variable qui va être créée
name	nom du bean qui va fournir la valeur
property	propriété du bean qui va fournir la valeur
scope	portée du bean
toScope	portée de la variable créée
type	type de la variable créée
value	

Exemple :

```
<jsp:useBean id="utilisateur" scope="page" class=" com.jmd.test.struts.data.Utilisateur"/>
<bean:define id="nomUtilisateur" name="utilisateur" property="nom"/>
Bienvenue <%= nomUtilisateur %>
```

Cet exemple permet de définir un bean de type Utilisateur qui est stocké dans la portée page. Une variable nomUtilisateur est définie et initialisée avec la valeur de la propriété nom du bean de type Utilisateur.

58.4.2.3. Le tag <bean:header>

Le tag <bean:header> est similaire au tag <bean:cookie> mais il permet de manipuler des données contenues dans l'en-tête de la requête HTTP.

58.4.2.4. Le tag <bean:include>

Le tag <bean:include> permet d'évaluer et d'inclure le rendu d'une autre page. Son mode de fonctionnement est similaire au tag JSP <jsp:include> excepté que le rendu de la page n'est pas inclus directement dans la page mais dans une variable.

Il possède plusieurs attributs :

Attribut	Rôle
id	nom de la variable créée qui va contenir le résultat de la page. Cette variable sera stockée dans la portée page.
forward	nom d'une redirection globale définie dans le fichier de configuration
href	URL de la page
page	URI relative au contexte de l'application de la page

Exemple :

```
<bean:include id="barreNavigation"  page="/navigation.jsp"/>
<bean:write name="barreNavigation"  filter="false" />
```

Ce tag est utile notamment pour obtenir un document XML qu'il sera alors possible de manipuler.

58.4.2.5. Le tag <bean:message>

Le tag <bean:message> permet d'obtenir la valeur d'un libellé contenu dans un ResourceBundle.

Il possède plusieurs attributs :

Attribut	Rôle
arg0	valeur du premier paramètre de remplacement
arg1	valeur du second paramètre de remplacement
arg2	valeur du troisième paramètre de remplacement
arg3	valeur du quatrième paramètre de remplacement
arg4	valeur du cinquième paramètre de remplacement
bundle	nom du bean qui encapsule le ResourceBundle (utilisé lorsque que plusieurs ResourceBundles sont définis)
key	clé du libellé à obtenir
locale	nom du bean qui stocke la Locale dans la session
name	nom du bean qui encapsule les données
property	propriété du bean précisé par l'attribut name contenant la valeur du libellé
scope	portée du bean précisé par l'attribut name

58.4.2.6. Le tag <bean:page>

Le tag <bean:page> permet d'obtenir une variable implicite définie par l'API JSP contenue dans la portée page.

Il possède plusieurs attributs :

Attribut	Rôle
id	nom de la variable à créer
property	variable implicite à extraire. Les valeurs possibles sont : application, config, request, response ou session

58.4.2.7. Le tag <bean:param>

Le tag <bean:param> est similaire au tag <bean:cookie> mais il permet de manipuler des données contenues dans les paramètres de la requête HTTP.

58.4.2.8. Le tag <bean:resource>

Le tag <bean:resource> permet d'obtenir la valeur d'une ressource sous la forme d'un objet de type java.io.InputStream ou String.

Il possède plusieurs attributs :

Attribut	Rôle
id	nom de la variable à créer
name	URI de la ressource relative à l'application à utiliser
input	permet d'obtenir la ressource sous la forme d'un objet de type java.io.InputStream. Sinon c'est un objet de type String qui est retourné

58.4.2.9. Le tag <bean:size>

Le tag <bean:size> permet d'obtenir le nombre d'éléments d'une collection ou d'un tableau. Ce tag créé une variable de type java.lang.Integer.

Il possède plusieurs attributs :

Attribut	Rôle
id	nom de la variable à créer
collection	expression renvoyant la collection à traiter
name	nom du bean qui encapsule la collection
property	propriété du bean qui encapsule la collection
scope	portée du bean

Exemple :

```
<bean:size id="count" name="elements" />
```

58.4.2.10. Le tag <bean:struts>

Le tag <bean:struts> permet de copier un objet Struts (FormBean, Mapping, Forward) dans une variable.

Il possède plusieurs attributs :

Attribut	Rôle
id	nom de la variable à créer (attribut obligatoire)
formBean	nom du bean de type ActionForm
forward	nom de l'objet global de type ActionForward
mapping	nom de l'objet de type ActionMapping

58.4.2.11. Le tag <bean:write>

Le tag <bean:write> permet d'envoyer dans le JspWrite courant la valeur d'un bean ou d'une propriété d'un bean.

Il possède plusieurs attributs :

Attribut	Rôle

bundle	nom du bean qui encapsule le ResourceBundle (utilisé lorsque plusieurs ResourceBundles sont définis)
filter	la valeur true permet de remplacer les caractères spécifiques d'HTML par leur entité correspondante
format	format de conversion en chaîne de caractères
formatKey	clé du ResourceBundle qui précise le format de conversion en chaîne de caractères
ignore	la valeur true permet d'ignorer l'inexistence du bean dans la portée. La valeur false lève une exception si le bean n'est pas trouvé dans la portée. Par défaut, false
locale	nom du bean qui stocke la Locale dans la session
name	nom du bean qui encapsule les données (attribut obligatoire)
property	propriété du bean
scope	portée du bean

Exemple :

```
<jsp:useBean id="utilisateur" scope="page" class=" com.jmd.test.struts.data.Utilisateur"/>
<bean:write name="utilisateur" property="nom"/>
```

L'attribut format du tag permet de formater les données restituées par le bean.

Exemple :

```
<p><bean:write  name="monbean" property="date" format="dd/MM/yyyy  HH:mm"/></p>
```

L'attribut formatKey du tag permet de formater les données restituées par le bean à partir d'une clé des ResourceBundle : ceci permet d'internationaliser le formatage.

Exemple :

```
<p><bean:write  name="monbean" property="date"  formatKey="date.format"/></p>
```

Dans le fichier ApplicationResources.properties

date.format=dd/MM/yyyy HH:mm

Dans le fichier ApplicationResources_en.properties

date.format=MM/dd/yyyy HH:mm

Il est important que le format précisé soit compatible avec la Locale courante sinon une exception est levée

Exemple :

```
javax.servlet.jsp.JspException: Wrong format string: '#.##0,00'
          at org.apache.struts.taglib.bean.WriteTag.formatValue(WriteTag.java:376)
          at org.apache.struts.taglib.bean.WriteTag.doStartTag(WriteTag.java:292)
```

58.4.3. La bibliothèque de tags Logic

Cette bibliothèque fournie des tags pour faciliter l'utilisation de logique de traitements pour l'affichage des pages.

Pour utiliser cette bibliothèque, il faut, comme pour toute bibliothèque de tags personnalisés, réaliser plusieurs opérations :

1. copier le fichier struts-logic.tld dans le répertoire WEB-INF de la webapp
2. configurer le fichier WEB-INF/web.xml pour déclarer la bibliothèque de tag

```
<taglib>
  <taglib-uri>struts-logic.tld</taglib-uri>
  <taglib-location>/WEB-INF/struts-logic.tld</taglib-location>
</taglib>
```

3. ajouter dans chaque page JSP qui va utiliser cette bibliothèque un tag de directive taglib précisant l'utilisation de la bibliothèque

```
<%@ taglib uri="struts-logic.tld" prefix="logic" %>
```

La plupart de ces tags encapsulent des tags de conditionnement des traitements ou d'exécution d'opérations sur le flot des traitements.

L'utilisation de ces tags évite l'utilisation de code Java dans les JSP.

Exemple :

```
  <jsp:useBean id="elements"  scope="request" class="java.util.List" />
    ...
  <%
    for (int i = 0; i  < elements.size(); i++)
    {
      MonElement monElement =  (MonElement)elements.get(i);
  %>
  <%=monElement.getLibelle()%>
  <%
    }
  %>
```

Tout le code Java peut être remplacé par l'utilisation de tag de la bibliothèque struts-logic.

Exemple :

```
<jsp:useBean  id="elements" scope="request" class="java.util.List" />
<logic:iterate id="monElement" name="elements" type="com.jmd.test.struts.data..MonElement">
  <bean:write name="monElement" property="libelle"/>
</logic:iterate>
```

Cette bibliothèque définit une quinzaine de tags.

Dans différents exemples de cette section, le bean suivant sera utilisé

Exemple :

```
package test.struts.data;

import java.util.Date;

public  class MonBean {
  private String libelle;
  private Integer valeur;
  private Date date;

  public MonBean() {
    libelle="libelle de test";
    valeur = new Integer(123456);
    date = new Date();
  }

  public void setLibelle(String libelle) {
    this.libelle = libelle;
  }
```

```
  public String getLibelle() {
    return libelle;
  }

  public void setDate(Date date) {
    this.date = date;
  }

  public Date getDate() {
    return date;
  }

  public void  setValeur(Integer valeur) {
    this.valeur =  valeur;
  }

  public Integer  getValeur() {
    return valeur;
  }
}
```

L'intérêt de cette bibliothèque a largement diminué depuis le développement de la JSTL qui intègre en standard des fonctionnalités équivalentes. Il est d'ailleurs fortement recommandé d'utiliser dès que possible les tags de la JSTL à la place des tags de Struts.

58.4.3.1. Les tags <logic:empty> et <logic:notEmpty>

Le tag <logic:empty> permet de tester si une variable est null ou vide. Le tag <logic:notEmpty> permet de faire le test opposé.

Il possède plusieurs attributs :

Attribut	Rôle
name	nom de la variable à tester si l'attribut property n'est pas précisé sinon c'est le nom de l'entité à tester (attribut obligatoire)
property	Nom de la propriété de la variable à tester
scope	Portée contenant la variable

58.4.3.2. Les tags <logic:equal> et <logic:notEqual>

Le tag <logic:equal> permet de tester l'égalité entre une variable et une valeur. Le tag <logic:notEqual> permet de faire le test opposé.

Ils possèdent plusieurs attributs :

Attribut	Rôle
Value	contient la valeur : celle-ci peut être une constante ou déterminée dynamiquement par exemple avec le tag JSP <%= ... %> (attribut obligatoire)
cookie	nom du cookie dont la valeur doit être testée
header	nom de l'attribut de l'en-tête http dont la valeur doit être testée
name	nom de la variable dont la valeur doit être testée
property	nom de la propriété de la variable à tester

parameter	nom du paramètre http dont la valeur doit être testée
scope	portée contenant la variable

Exemple :

```
   <% int valeurReference = 123456; %>
 ...
   <logic:equal name="monbean"
   property="valeur"
   value="<%= valeurReference %>">
   <p>La valeur est égale</p>
   </logic:equal>
```

58.4.3.3. Les tags <logic:lessEqual>, <logic:lessThan>, <logic:greaterEqual>, et <logic:greaterThan>

Ils sont similaires au tag <logic:equal> mais permettent respectivement de tester les conditions inférieur ou égal, strictement inférieur, supérieur ou égal et strictement supérieur.

58.4.3.4. Les tags <logic:match> et <logic:notMatch>

Le tag <logic:match> permet de tester si une valeur est contenue dans une variable. Le tag <logic:notMatch> permet de faire le test opposé.

Ils possèdent plusieurs attributs :

Attribut	Rôle
value	contient la valeur : celle-ci peut être une constante ou déterminée dynamiquement par exemple avec le tag JSP <%= ... %> (attribut obligatoire)
cookie	nom du cookie dont la valeur doit être testée
header	nom de l'attribut de l'en-tête http dont la valeur doit être testée
name	nom de la variable dont la valeur doit être testée
property	nom de la propriété de la variable à tester
parameter	nom du paramètre http dont la valeur doit être testée
scope	portée contenant la variable
location	permet de préciser la localisation de la valeur à rechercher dans la variable. Les valeurs possibles sont start et end pour une recherche respectivement en début et en fin. Sans préciser cet attribut, la recherche se fait dans toute la variable

58.4.3.5. Les tags <logic:present> et <logic:notPresent>

Le tag <logic:present> permet de tester l'existence d'une entité dans une portée donnée. Le tag <logic:notPresent> permet de faire le test opposé.

Ils possèdent plusieurs attributs :

Attribut	Rôle
cookie	nom du cookie dont la valeur doit être testée

header	nom de l'attribut de l'en-tête http dont la valeur doit être testée
name	nom de la variable dont la valeur doit être testée
property	nom de la propriété de la variable à tester
parameter	nom du paramètre http dont la valeur doit être testée
scope	portée contenant la variable

58.4.3.6. Le tag <logic:forward>

Le tag <logic:forward> permet de transférer le traitement de la requête vers une page définie dans les redirections globales de l'application.

Il ne possède qu'un seul attribut name qui permet de préciser le nom de la redirection globale définie dans le fichier de configuration struts-config.xml

Exemple : dans une JSP

```
<logic:forward name=">strong<login>/strong<" />
```

Exemple : dans le fichier de configuration

```
  <global-forwards>
    <forward name=">strong<login>/strong<" path="/login.jsp"/>
  </global-forwards>
```

58.4.3.7. Le tag <logic:redirect>

Le tag <logic:redirect> permet de rediriger l'affichage vers une autre page en utilisant la méthode HttpServletResponse.sendRedirect().

Il possède plusieurs attributs :

Attribut	Rôle
forward	nom de la redirection globale définie dans le fichier de configuration struts-config.xml à utiliser
href	URL de la ressource à utiliser
page	URL de la ressource relative au contexte de l'application (doit obligatoirement commencer par /)
name	collection de type Map qui contient les paramètres à passer à la ressource
paramId	nom de l'unique paramètre passé à la ressource
paramName	nom d'une variable dont la valeur sera utilisée comme valeur du paramètre
paramProperty	propriété de la variable paramName dont la valeur sera utilisée comme valeur du paramètre

L'avantage de ce tag est de permettre de modifier les paramètres fournis à la ressource.

58.4.3.8. Le tag <logic:iterate>

Ce tag permet de réaliser une itération sur une collection d'objets. Le corps du tag sera évalué pour chaque occurrence de l'itération.

Il possède plusieurs attributs :

Attribut	Rôle
id	nom de la variable qui va contenir l'occurrence courante de l'itération (attribut obligatoire)
name	nom de la variable qui contient la collection à parcourir
property	nom de la propriété de la variable name qui contient la collection à parcourir
scope	portée de la variable qui contient la collection
type	type pleinement qualifié des occurrences de la collection
indexId	nom de la variable qui va contenir l'index de l'occurrence courante
length	nombre maximum d'occurrences à traiter. Par défaut toute la collection est parcourue
offset	index de la première occurrence de l'itération. Par défaut c'est la première occurrence de la collection

58.5. La validation de données

La méthode validate() de la classe ActionForm permet de réaliser une validation des données fournies dans la requête.

Elle est appelée par l'ActionServlet lorsque l'attribut validate est positionné à true dans le tag <action>.

Exemple :

Exemple :

```
<action  path="/validerproduit"
         type="test.struts.controleur.ValiderProduitAction"
        name="saisirProduitForm"
        validate="true">
  <forward name="succes"  path="/listeproduit.jsp"/>
  <forward name="echec"  path="/saisirproduit.jsp"/>
</action>
```

Pour définir ses propres validations, il faut redéfinir la méthode validate() pour y coder les règles de validation. Si une erreur est détectée lors de l'exécution de ces règles, il faut instancier un objet de type ActionError et l'ajouter à l'objet ActionErrors retourné par la méthode validate(). Cet ajout se fait en utilisant la méthode add().

58.5.1. La classe ActionError

Cette classe encapsule une erreur survenue lors de la validation des données. C'est dans la méthode validate() de la classe ActionForm que les traitements doivent créer des instances de cette classe.

Le constructeur de cette classe attend en paramètre une chaîne de caractères qui précise le nom d'une clé du message d'erreur correspondant au message de l'erreur défini dans le fichier ressource bundle de l'application.

La méthode validate() de la classe ActionForm possède deux surcharges :

public ActionErrors validate(ActionMapping mapping, javax.servlet.http.HttpServletRequest request)

public ActionErrors validate(ActionMapping mapping, javax.servlet.ServletRequest request)

La première version est essentiellement mise en oeuvre car elle est utilisée pour les applications web.

Elle renvoie un objet de type ActionErrors qui va contenir les éventuelles erreurs détectées lors de la validation. Si la collection est vide ou nulle cela précise que la validation a réussie. Ceci permet à l'ActionServlet de savoir si elle va pouvoir appeler la méthode execute() de l'Action.

Par défaut, la méthode validate() de la classe ActionForm renvoie systématiquement null. Il est donc nécessaire de sous classer la classe ActionForm et de redéfinir la méthode validate().

Exemple :

Exemple :

```
public ActionErrors validate(ActionMapping  mapping, HttpServletRequest request) {
  ActionErrors errors = new ActionErrors();
  if ((nomUtilisateur == null) ||  (nomUtilisateur.length() == 0))
    errors.add("nomUtilisateur",  new ActionError("erreur.nomutilisateur.obligatoire"));
  if ((mdpUtilisateur == null) ||  (mdpUtilisateur.length() == 0))
    errors.add("mdpUtilisateur",  new ActionError("erreur.mdputilisateur.obligatoire"));
  return errors;
}
```

Ce mécanisme peut aussi être mis en oeuvre dans la méthode execute() de la classe Action.

58.5.2. La classe ActionErrors

Cette classe encapsule une collection de type HashMap d'objets ActionError générés lors d'une validation.

C'est la méthode validate() de la classe ActionForm qui renvoie une instance de cette classe. Les traitements qu'elle contient se chargent de créer une instance de cette classe et d'utiliser la méthode add() pour ajouter des instances de la classe ActionError pour chaque erreur rencontrées.

Il est aussi possible de définir des erreurs dans la méthode : il faut créer un objet de type ActionErrors, utiliser sa méthode add() pour chaque erreur à ajouter et appeler la méthode saveErrors de la classe Action pour sauvegarder les erreurs.

Exemple :

```
  public ActionForward execute(
    ActionMapping       mapping,
    ActionForm          form,
    HttpServletRequest  request,
    HttpServletResponse response) throws  Exception
  {
    DynaActionForm daf       = (DynaActionForm) form;
    ActionForward  resultat  =  mapping.findForward("succes");
    String         reference = (String)  daf.get("reference");
    String         libelle   = (String) daf.get("libelle");
    int            prix      = Integer.parseInt((String)  daf.get("prix"));

    System.out.println("reference=" +  reference);
    System.out.println("libelle=" +  libelle);
    System.out.println("prix=" +  prix);

    if ((reference == null) ||  (reference.equals(""))) {
      ActionErrors errors = new ActionErrors();
      errors.add("reference", new  ActionError("app.saisirproduit.erreur.reference"));
      saveErrors(request, errors);

      resultat =  mapping.findForward("echec");
    }
    return resultat;
  }
```

Remarque : dans cet exemple, la validation des données est effectuée dans la méthode execute. Il est préférable d'effectuer cette tâche via une des fonctionnalités proposées par Struts (validation via l'ActionForm ou le plug-in Validator).

58.5.3. L'affichage des messages d'erreur

Le tag <html:errors> permet d'afficher les erreurs contenues dans l'instance courante de la classe ActionErrors.

Exemple :

```
<html:errors/>
```

Le plus simple est d'utiliser ce tag en début du corps de la page. Il se charge d'afficher toutes les erreurs (les erreurs globales et celles dédiées à un élément du formulaire) pour permettre leur gestion de façon globale à toute la page.

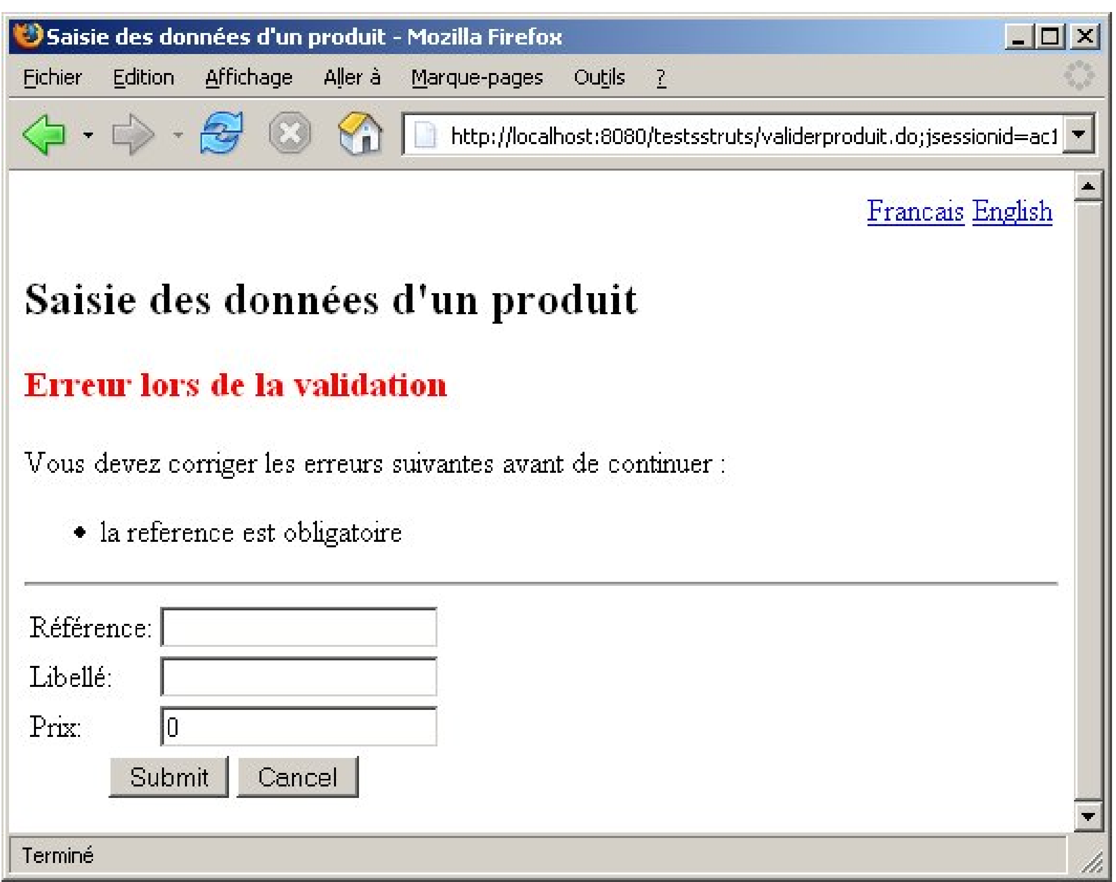
Ce tag recherche dans les ResourceBundles les deux clés errors.header et errors.footer dont la valeur sera affichée avant les messages. A partir de la version 1.1 de Struts, les clés errors.prefix et error.suffix sont recherchées dans les ResourceBundles et ajoutées respectivement avant et après chaque message.

Exemple :

```
errors.prefix=<li>
errors.suffix=</li>
errors.header=<h3><font  color\="red">Erreur lors de la  validation</font></h3>
Vous devez corriger les erreurs suivantes avant de continuer \:<ul>
errors.footer=</ul><hr>
```

L'utilisation de tags HTML dans les ResourceBundles peut paraître choquante mais c'est la solution utilisée par Struts.

Exemple :



Avec Struts 1.1, il est aussi possible d'utiliser le tag <html:errors> pour afficher des messages d'erreurs liés à un composant du formulaire. Dans ce cas, l'approche est légèrement différente.

L'exemple ci-dessous va afficher un message personnalisé pour un composant et un message d'erreur général.

Exemple : ApplicationResources.properties

```
...
app.saisirproduit.erreur.reference=la reference saisie est erronée
app.saisirproduit.erreur.libelle=le libelle saisie est erronée
app.saisirproduit.erreur.globale=une ou plusieurs erreurs sont survenues

errors.prefix=
errors.suffix=
errors.header=
errors.footer=
...
```

Comme les clés préfixées par errors sont utilisées pour chaque affichage d'erreur, leur contenu est laissé vide.

L'action instancie des objets de type ActionError si une erreur est détectée sur les données et l'associe au composant correspondant. Lors de l'ajout d'une erreur, il faut préciser l'identifiant du composant correspondant à son attribut property dans le tag de la page.

Si au moins une erreur est détectée sur une donnée alors une erreur globale est ajoutée à la liste des erreurs. Pour cela, il faut utiliser la constante ActionErrors.GLOBAL_ERROR lors de l'ajout de l'erreur dans la collection ActionErrors.

Exemple Struts 1.1 :

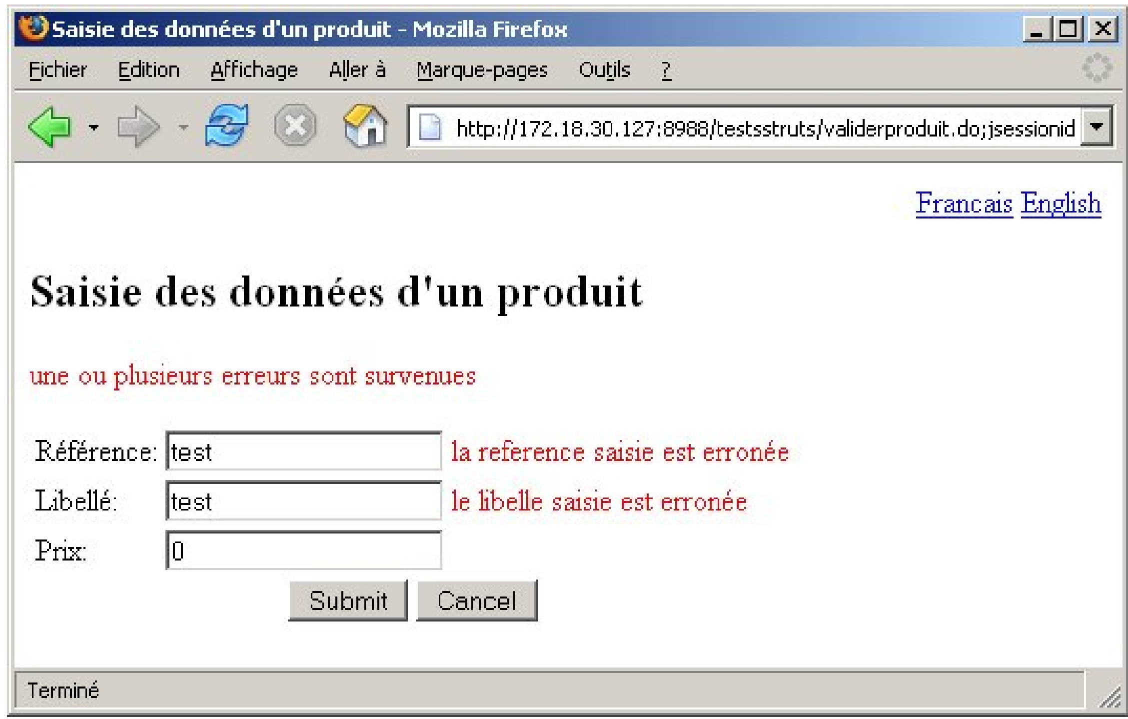
```
  ...
  public  ActionForward execute(
     ActionMapping       mapping,
     ActionForm          form,
     HttpServletRequest  request,
     HttpServletResponse response) throws Exception
  {
    DynaActionForm daf       = (DynaActionForm) form;
    ActionForward  resultat  = mapping.findForward("succes");
    ActionErrors   errors    = new ActionErrors();
    String         reference = (String) daf.get("reference");
    String         libelle   = (String) daf.get("libelle");
    int            prix      = Integer.parseInt((String)  daf.get("prix"));

    if (reference.equals("test")) {
      errors.add("reference", new  ActionError("app.saisirproduit.erreur.reference"));
    }
    if (libelle.equals("test")) {
      errors.add("libelle", new  ActionError("app.saisirproduit.erreur.libelle"));
    }

    if (!errors.isEmpty())
    {
      errors.add(ActionErrors.GLOBAL_ERROR,
        new  ActionError("app.saisirproduit.erreur.globale"));
      saveErrors(request, errors);
      resultat = mapping.findForward("echec");
    }

    return resultat;
  }
  ...
```

Il ne reste plus qu'à assurer l'affichage des messages d'erreurs dans la page. Pour le message associé à un composant il faut utiliser l'attribut property du tag <html:errors> en précisant comme valeur le nom du composant dont les messages doivent être affichés.

Pour afficher les messages d'erreurs globaux, il faut préciser dans l'attribut property la valeur de la constante ActionErrors.GLOBAL_ERROR.

Exemple Struts 1.1 :

```
<%@ taglib  uri="/WEB-INF/struts-bean.tld" prefix="bean"%>
<%@ taglib  uri="/WEB-INF/struts-html.tld" prefix="html"%>
<%@ taglib  uri="/WEB-INF/struts-logic.tld" prefix="logic"%>
<%@ page  contentType="text/html;charset=windows-1252"%>
<%@ page  import  ="org.apache.struts.action.*"    %>
<html:html locale="true">
  <head>
    <meta http-equiv="Content-Type" content="text/html;  charset=windows-1252"/>
    <title>
      <bean:message key="app.saisirproduit.titre"/>
    </title>
  </head>
  <body>
    <table width="100%">
      <tr>
        <td align="right">
          <html:link  href="changerlangue.do?langue=fr">Francais</html:link>
          <html:link  href="changerlangue.do?langue=en">English</html:link>
        </td>
      </tr>
    </table>
    <h2>
      <bean:message key="app.saisirproduit.titre"/>
    </h2>
    <html:form action="validerproduit.do"  focusIndex="reference">
      <logic:present name="<%=Action.ERROR_KEY%>">
      <P style="color:red;"><html:errors  property="<%=ActionErrors.GLOBAL_ERROR%>"/></P>
      </logic:present>
       <table>
        <tr>
          <td>
            <bean:message  key="app.saisirproduit.libelle.reference"/>:
         </td>
          <td>
            <html:text property="reference"/>
          </td>
          <td style="color:red;"><html:errors  property="reference"/></td>
        </tr>
        <tr>
          <td>
            <bean:message key="app.saisirproduit.libelle.libelle"/>:
          </td>
          <td>
            <html:text property="libelle"/>
          </td>
          <td style="color:red;"><html:errors  property="libelle"/></td>
        </tr>
        <tr>
          <td>
            <bean:message key="app.saisirproduit.libelle.prix"/>:
          </td>
          <td>
            <html:text property="prix"/>
          </td>
          <td></td>
        </tr>
        <tr>
          <td colspan="3" align="center">
            <html:submit/>
            <html:cancel/>
          </td>
        </tr>
      </table>
    </html:form>
  </body>
</html:html>
```



58.5.4. Les classes ActionMessage et ActionMessages

La classe ActionMessage, apparue avec Struts 1.1, fonctionne de la même façon que la classe ActionError mais elle encapsule des messages d'information qui ne sont pas des erreurs.

Ce type de message est pratique notamment pour afficher des messages de confirmation ou d'information aux utilisateurs.

La classe ActionMessages encapsule une collection d'ActionMessage.

Exemple :

```
ActionMessages  actionMessages = new ActionMessages();
actionMessages.add(ActionMessages.GLOBAL_MESSAGE,
  new  ActionMessage("liste.incomplete"));
saveMessages(request,actionMessages);
```

La méthode add() permet d'ajouter des messages dans la collection.

La méthode clear() permet de supprimer tous les messages de la collections.

La méthode isEmpty() permet de savoir si la collection est vide et la méthode size() permet de connaître le nombre de messages stockés dans la collection.

58.5.5. L'affichage des messages

Le tag <html:messages> permet d'afficher les messages contenus dans l'instance courante de la classe ActionMessages.

Exemple :

```
<logic:messagesPresent message="true">
 <html:messages  id="message" message="true">
   <bean:write  name="message"/>
```

```
    </html:messages>
  </logic:messagesPresent>
```

La suite de ce chapitre sera développé dans une version future de ce document

59. JSF (Java Server Faces)

59.1. La présentationde JSF

Les technologies permettant de développer des applications web avec Java ne cessent d'évoluer :

1. Servlets
2. JSP
3. MVC Model 1 : servlets + JSP
4. MVC Model 2 : un seule servlet + JSP
5. Java Server Faces

Java Server Faces (JSF) est une technologie dont le but est de proposer un framework qui facilite et standardise le développement d'applications web avec Java. Son développement a tenu compte des différentes expériences acquises lors de l'utilisation des technologies standards pour le développement d'applications web (servlet, JSP, JSTL) et de différents frameworks (Struts, ...).

Le grand intérêt de JSF est de proposer un framework qui puisse être mis en oeuvre par des outils pour permettre un développement de type RAD pour les applications web et ainsi faciliter le développement des applications de ce type. Ce type de développement était déjà courant pour des applications standalone ou client/serveur lourd avec des outils tel que Delphi de Borland, Visual Basic de Microsoft ou Swing avec Java.

Ce concept n'est pourtant pas nouveau dans les applications web puisqu'il est déjà mis en oeuvre par WebObject d'Apple et plus récemment par ASP.Net de Microsoft mais sa mise en oeuvre à grande échelle fût relativement tardive. L'adoption du RAD pour le développement web trouve notamment sa justification dans le coût élevé de développement de l'IHM à la « main » et souvent par copier/coller d'un mixe de plusieurs technologies (HTML, Javascript, ...), rendant fastidieux et peu fiable le développement de ces applications.

Plusieurs outils commerciaux intègrent déjà l'utilisation de JSF notamment Studio Creator de Sun, WSAD d'IBM, JBuilder de Borland, JDevelopper d'Oracle, ...

Même si JSF peut être utilisé par codage à la main, l'utilisation d'un outil est fortement recommandée pour pouvoir mettre en oeuvre rapidement toute la puissance de JSF.

Ainsi de par sa complexité et sa puissance, JSF s'adapte parfaitement au développement d'applications web complexes en facilitant leur écriture.

Les pages officielles de cette technologie sont à l'url : http://java.sun.com/j2ee/javaserverfaces/

La version 1.0 de Java Server Faces, développée sous la JSR-127, a été validée en mars 2004.

JSF est une technologie utilisée côté serveur dont le but est de faciliter le développement de l'interface utilisateur en séparant clairement la partie « interface » de la partie « métier » d'autant que la partie interface n'est souvent pas la plus compliquée mais la plus fastidieuse à réaliser.

Cette séparation avait déjà été initiée avec la technologie JSP et particulièrement les bibliothèques de tags personnalisés. Mais JSF va encore plus loin en reposant sur le modèle MVC et en proposant de mettre en oeuvre :

- l'assemblage de composants serveur qui génèrent le code de leur rendu avec la possibilité d'associer certains

composants à une source de données encapsulée dans un bean
* l'utilisation d'un modèle de développement standardisé reposant sur l'utilisation d'événements et de listener
* la conversion et la validation des données avant leur utilisation dans les traitements
* la gestion de l'état des composants de l'interface graphique
* la possibilité d'étendre les différents modèles et de créer ces propres composants
* la configuration de la navigation entre les pages
* le support de l'internationalisation
* le support pour l'utilisation par des outils graphiques du framework afin de faciliter sa mise en oeuvre

JSF se compose :

* d'une spécification qui définit le mode de fonctionnement du framework et une API : l'ensemble des classes de l'API est contenu dans les packages javax.faces.
* d'une implémentation de référence
* de bibliothèques de tags personnalisés fournies par l'implémentation pour utiliser les composants dans les JSP, gérer les événements, valider les données saisies, ...

Le rendu des composants ne se limite pas à une seule technologie même si l'implémentation de référence ne propose qu'un rendu des composants en HTML.

Le traitement d'une requête traitée par une application utilisant JSF utilise un cycle de vie particulier constitué de plusieurs étapes :

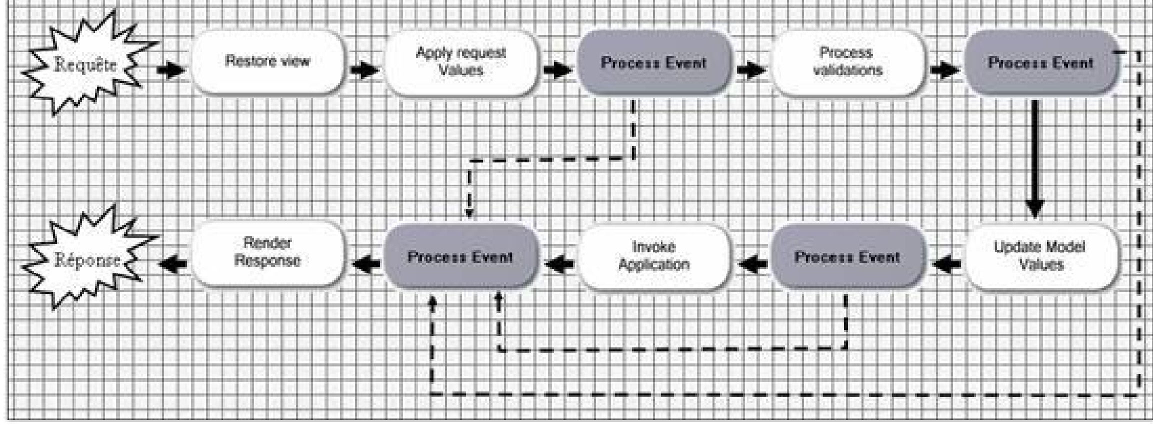
* Création de l'arbre de composants
* Extraction des données des différents composants de la page
* Conversion et validation des données
* Extraction des données validées et mise à jour du modèle de données (javabean)
* Traitements des événements liés à la page
* Génération du rendu de la réponse

Ces différentes étapes sont transparentes lors d'une utilisation standard de JSF.

59.2. Le cycle de vie d'une requête

JSF utilise la notion de vue (view) qui est composée d'une arborescence ordonnée de composants inclus dans la page.

Les requêtes sont prises en charge et gérées par le contrôleur d'une application JSF (en général une servlet). Celle ci va assurer la mise en oeuvre d'un cycle de vie des traitements permettant de traiter la requête en vue d'envoyer une réponse au client.



JSF propose pour chaque page un cycle de vie pour traiter la requête HTTP et générer la réponse. Ce cycle de vie est composé de plusieurs étapes :

1. Restore view ou Reconstruct Component Tree : cette première phase permet au serveur de recréer l'arborescence des composants qui composent la page. Cette arborescence est stockée dans un objet de type FacesContext et

sera utilisée tout au long du traitement de la requête.

2. Apply Request Value : dans cette étape, les valeurs des données sont extraites de la requête HTTP pour chaque composant et sont stockées dans leur composant respectif dans le FaceContext. Durant cette phase des opérations de conversions sont réalisées pour permettre de transformer les valeurs stockées sous forme de chaîne de caractères dans la requête http en un type utilisé pour le stockage des données.
3. Perform validations : une fois les données extraites et converties, il est possible de procéder à leur validation en appliquant les validators enregistrés auprès de chaque composant. Les éventuelles erreurs de conversions sont stockées dans le FaceContext. Dans ce cas, l'étape suivante est directement « Render Response » pour permettre de réafficher la page avec les valeurs saisies et afficher les erreurs
4. Synchronize Model ou update model values : cette étape permet de stocker dans les composants du FaceContext leur valeur locale validée respective. Les éventuelles erreurs de conversions sont stockées dans le FaceContext. Dans ce cas, l'étape suivante est directement « Render Response » pour permettre de réafficher la page avec les valeurs saisies et afficher les erreurs
5. Invoke Application Logic : dans cette étape, le ou les événements émis dans la page sont traités. Cette phase doit permettre de déterminer quelle sera la page résultat qui sera renvoyée dans la réponse en utilisant les règles de navigation définie dans l'application. L'arborescence des composants de cette page est créée.
6. Render Response : cette étape se charge de créer le rendue de la page de la réponse.

59.3. Les implémentations

Java Server Faces est une spécification : il est donc nécessaire d'obtenir une implémentation de la part d'un tiers.

Plusieurs implémentations commerciales ou libres sont disponibles, notamment l'implémentation de référence de Sun et MyFaces qui est devenu un projet du groupe Apache.

59.3.1. L'implémentation de référence

Comme pour toute JSR validée, Sun propose une implémentation de référence des spécifications de la JSR, qui soit la plus complète possible.

Plusieurs versions de l'implémentation de référence de Sun sont proposées :

Version	Date de diffusion
1.0	Mars 2004
1.1	Mai 2004
1.1_01	Septembre 2004

La solution la plus simple pour utiliser l'implémentation de référence est d'installer le JWSDK 1.3 qui est fourni en standard avec l'implémentation de référence de JSF. La version de JSF fournie avec le JWSDK 1.3 est la 1.0.

Pour utiliser la version 1.1, il faut supprimer le répertoire jsf dans le répertoire d'installation de JWSDK, télécharger l'implémentation de référence, décompresser son contenu dans le répertoire d'installation de JWSDK et renommer le répertoire jsf-1_1_01 en jsf.

Il est aussi possible de télécharger l'implémentation de référence sur le site de Sun et de l'installer « manuellement » dans un conteneur web tel que Tomcat. Cette procédure sera détaillée dans une des sections suivantes.

Pour cela, il faut télécharger le fichier jsf-1_1_01.zip et le décompresser dans un répertoire du système. L'archive contient les bibliothèques de l'implémentation, la documentation des API et des exemples.

Les exemples de ce chapitre vont utiliser cette version 1.1 de l'implémentation de référence des JSF.

59.3.2. MyFaces

MyFaces est une implémentation libre des Java Server Faces qui est devenu un projet du groupe Apache.

Il propose en plus plusieurs composants spécifiques en plus de ceux imposés par les spécifications JSF.

Le site de MyFaces est à l'url : http://myfaces.apache.org/

Il faut télécharger le fichier et le décompresser dans un répertoire du système. Il suffit alors de copier le fichier myfaces-examples.war dans le répertoire webapps de Tomcat. Relancez Tomcat et saisie l'url http://localhost:8080/myfaces-examples

Pour utiliser MyFaces dans ses propres applications, il faut réaliser plusieurs opérations.

Il faut copier les fichiers *.jar du répertoire lib de MyFaces et myfaces-jsf-api.jar dans le répertoire WEB-INF/lib de la webapp.

Dans chaque page qui va utiliser les composants de MyFaces, il faut déclarer la bibliothèque de tags dédiés.

Exemple :

```
<%@ taglib uri="http://myfaces.sourceforge.net/tld/myfaces_ext_0_9.tld" prefix="x"%>
```

59.4. Le contenu d'une application

Les applications utilisant JSF sont des applications web qui doivent respecter les spécifications de J2EE.

En tant que telle, elles doivent avoir la structure définie par J2EE pour toutes les applications web :

/
/WEB-INF
/WEB-INF/web.xml

/WEB-INF/lib
/WEB-INF/classes

Le fichier web.xml doit contenir au minimum certaines informations notamment, la servlet faisant office de contrôleur, le mapping des url pour cette servlet et des paramètres.

Exemple :

```
<!DOCTYPE web-app PUBLIC
"-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <display-name>Test JSF</display-name>
  <description>Application de tests avec JSF</description>
  <context-param>
    <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
    <param-value>client</param-value>
  </context-param>
  <!-- Faces Servlet -->
  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup> 1 </load-on-startup>
  </servlet>
  <!-- Faces Servlet Mapping -->
  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.jsf</url-pattern>
  </servlet-mapping>
</web-app>
```

Chaque implémentation nécessite un certain nombre de bibliothèques tiers pour leur bon fonctionnement.

Par exemple, pour l'implémentation de référence, les bibliothèques suivantes sont nécessaires :

jsf-api.jar
jsf-ri.jar
jstl.jar
standard.jar
common-beanutils.jar
commons-digester.jar
commons-collections.jar
commons-logging.jar

Remarque : avec l'implémentation de référence, il n'y a aucun fichier .tld à copier car ils sont intégrés dans le fichier jsf-impl.jar.

Les fichiers nécessaires dépendent de l'implémentation utilisée.

Ces bibliothèques peuvent être mises à disposition de l'application selon plusieurs modes :

- incorporées dans le package de l'application dans le répertoire /WEB-INF/lib
- incluses dans le répertoire des bibliothèques partagées par les applications web des conteneurs web s'ils proposent une telle fonctionnalité. Par exemple avec Tomcat, il est possible de copier ces bibliothèques dans le répertoire shared/lib.

L'avantage de la première solution est de faciliter la portabilité de l'application sur différents conteneur web mais elle duplique ces fichiers si plusieurs applications utilisent JSF.

Les avantages et inconvénients de la première solution sont exactement l'opposé de la seconde solution. Le choix de l'une ou l'autre est donc à faire en fonction du contexte de déploiement.

59.5. La configuration de l'application

Toute application utilisant JSF doit posséder au moins deux fichiers de configuration qui vont contenir les informations nécessaires à la bonne configuration et exécution de l'application.

Le premier fichier est le descripteur de toute application web J2EE : le fichier web.xml contenu dans le répertoire WEB-INF.

Le second fichier est un fichier de configuration particulier au paramétrage de JSF au format XML nommé faces-config.xml.

59.5.1. Le fichier web.xml

Le fichier web.xml doit contenir au minimum certaines informations notamment, la servlet faisant office de contrôleur, le mapping des urls pour cette servlet et des paramètres pour configurer JSF.

Exemple :

```
<!DOCTYPE web-app PUBLIC
"-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <display-name>Test JSF</display-name>
  <description>Application de tests avec JSF</description>
  <context-param>
    <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
    <param-value>client</param-value>
  </context-param>

  <!-- Servlet faisant office de controleur-->
  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup> 1 </load-on-startup>
  </servlet>

  <!-Le mapping de la servlet -->
  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.faces</url-pattern>
  </servlet-mapping>
</web-app>
```

Le tag <servlet> permet de définir une servlet et plus particulièrement dans ce cas de préciser la servlet qui sera utilisée comme contrôleur dans l'application. Le plus simple est d'utiliser la servlet fournie avec l'implémentation de référence javax.faces.webapp.FacesServlet. Le tag <load-on-startup> avec comme valeur 1 permet de demander le chargement de cette servlet au lancement de l'application.

Le tag <servlet-mapping> permet de préciser le mapping des urls qui seront traitées par la servlet. Ce mapping peut prendre deux formes :

- mapping par rapport à une extension : exemple <url-pattern>*.faces</url-pattern>.
- mapping par rapport à un préfixe : exemple <url-pattern>/faces/*</url-pattern>.

Les URL utilisées pour des pages mettant en oeuvre JSF doivent obligatoirement passer par cette servlet. Ces urls peuvent être de deux formes selon le mapping défini.

Exemple :

- http://localhost:8080/nom_webapp/index.faces
- http://localhost:8080/nom_webapp/faces/index.jsp

Dans les deux cas, c'est la servlet utilisée comme contrôleur qui va déterminer le nom de la page JSP à utiliser.

Le paramètre de contexte javax.faces.STATE_SAVING_METHOD permet de préciser le mode d'échange de l'état de l'arbre des composants de la page. Deux valeurs sont possibles :

- client :
- server :

Il est possible d'utiliser l'extension .jsf pour les fichiers JSP utilisant JSF à condition de correctement configurer le fichier web.xml dans ce sens. Pour cela deux choses sont à faire :

- il faut demander le mapping des url terminant par .jsf par la servlet

  ```
  <servlet-mapping>
  <servlet-name>jsp</servlet-name>
  <url-pattern>*.jsf</url-pattern>
  </servlet-mapping>
  ```
- il faut préciser à la servlet le suffix par défaut à utiliser

  ```
  <context-param>
  <param-name>javax.faces.DEFAULT_SUFFIX</param-name>
  <param-value>.jsf</param-value>
  </context-param>
  ```

Le démarrage d'une application directement avec une page par défaut utilisant JSF ne fonctionne pas correctement. Il est préférable d'utiliser une page HTML qui va effectuer une redirection vers la page d'accueil de l'application

Exemple :

```
<html>
  <head>
    <meta http-equiv="Refresh" content= "0; URL=index.faces"/>
    <title>Demarrage de l'application</title>
  </head>
  <body>
    <p>D&eacute;marrage de l'application ...</p>
  </body>
</html>
```

Il suffit alors de préciser dans le fichier web.xml que cette page est la page par défaut de l'application.

Exemple :

```
...
<welcome-file-list>
  <welcome-file>index.htm</welcome-file>
</welcome-file-list>
...
```

59.5.2. Le fichier faces-config.xml

Le plus simple est de placer ce fichier dans le répertoire WEB-INF de l'application Web.

Il est aussi possible de préciser son emplacement dans un paramètre de contexte nommé javax.faces.application.CONFIG_FILES dans le fichier web.xml. Il est possible par ce biais de découper le fichier de configuration en plusieurs morceaux. Ceci est particulièrement intéressant pour de grosses applications car un seul fichier de configuration peut dans ce cas devenir très gros. Il suffit de préciser chacun des fichiers séparés par une virgule dans le tag <param-value>.

Exemple :

```
...
<context-param>
  <param-name>javax.faces.application.CONFIG_FILES</param-name>
  <param-value>
/WEB-INF/ma-faces-config.xml, /WEB-INF/navigation-faces.xml, /WEB-INF/beans-faces.xml
  </param-value>
</context-param>
...
```

Ce fichier au format XML permet de définir et de fournir des valeurs d'initialisation pour des ressources nécessaires à l'application utilisant JSF.

Ce fichier doit impérativement respecter la DTD proposée par les spécifications de JSF :

http://java.sun.com/dtd/web-facesconfig_1_0.dtd

Le tag racine du document XML est le tag <face-config>. Ce tag peut avoir plusieurs tags fils :

Tag	Rôle
application	permet de préciser ou de remplacer des éléments de l'application
factory	permet de remplacer des fabriques par des fabriques personnalisées de certaines ressources (FacesContextFactory, LifeCycleFactory, RenderKitFactory, ...)
component	définit un composant graphique personnalisé
converter	définit un convertisseur pour encoder/décoder les valeurs des composants graphiques (conversion de String en Object et vice et versa)
managed-bean	définit un objet utilisé par un composant qui est automatiquement créé, initialisé et stocké dans une portée précisée
navigation-rule	définit les règles qui permettent de déterminer l'enchaînement des traitements de l'application
referenced-bean	
render-kit	définit un kit pour le rendu des composants graphiques
lifecycle	
validator	définit un validateur personnalisé de données saisies dans un composant graphique

Ces tags fils peuvent être utilisé 0 ou plusieurs fois dans le tag <face-config>.

Le tag <application> permet de préciser des informations sur les entités utilisées par l'internationalisation et/ou de remplacer des éléments de l'application.

Les éléments à remplacer peuvent être : ActionListener, NavigationHandler, ViewHandler, PropertyResolver, VariableResolver. Ceci n'est utile que si la version fournie dans l'implémentation ne correspond pas aux besoins et doit être personnalisée par l'écriture d'une classe dédiée.

Le tag fils <message-bundle> permet de préciser le nom de base des fichiers de ressources utiles à l'internationalisation.

Le tag <locale-config> permet de préciser quelles sont les locales qui sont supportées par l'application. Il faut utiliser autant de tag fils <supported-locale> que de locales supportées. Le tag fil <default-locale> permet de préciser la locale par défaut.

Exemple :

```
...
<application>
  <message-bundle>com.jmdoudoux.test.jsf.monapp.bundles.Messages</message-bundle>
```

```
  <locale-config>
    <default-locale>fr</default-locale>
    <supported-locale>en</supported-locale>
  </locale-config>
</application>
...
```

59.6. Les beans

Les beans sont largement utilisées dans une application utilisant JSF notamment pour permettre l'échange de données entre les différentes entités et le traitement des événements.

Les beans sont des classes qui respectent une spécification particulière notamment la présence :

- de getters et de setters qui respectent une convention de nommage particulière pour les attributs
- un constructeur par défaut sans arguments

59.6.1. Les beans managés (managed bean)

Les beans managés sont des javabeans dont le cycle de vie va être géré par le framework JSF en fonction des besoins et du paramétrage fourni dans le fichier de configuration.

Dans le fichier de configuration, chacun de ces beans doit être déclaré avec un tag <managed-bean>. Ce tag possède trois tags fils obligatoires :

- <managed-bean-name> : le nom attribué au bean (celui qui sera utilisé lors de son utilisation)
- <managed-bean-class> : le type pleinement qualifié de la classe du bean
- <managed-bean-scope> : précise la portée dans laquelle le bean sera stockée et donc utilisable

La portée peut prendre les valeurs suivantes :

- request : cette portée est limitée entre l'émission de la requête et l'envoi de la réponse. Les données stockées dans cette portée sont utilisables lors d'un transfert vers une autre page (forward). Elles sont perdues lors d'une redirection (redirect).
- session : cette portée permet l'échange de données entre plusieurs échanges avec un même client
- application : cette portée permet l'accès à des données pour toutes les pages d'une même application quelque soit l'utilisateur

Exemple :

```
...
<managed-bean>
  <managed-bean-name>login</managed-bean-name>
  <managed-bean-class>com.jmd.test.jsf.LoginBean</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
...
```

Il est possible de fournir des valeurs par défaut aux propriétés en utilisant le tag <managed-property>. Ce tag possède deux tags fils :

- <property-name> : nom de la propriété du bean
- <value> : valeur à associer à la propriété

Exemple :

```
...
<managed-bean>
  <managed-bean-name>login</managed-bean-name>
  <managed-bean-class>com.jmd.test.jsf.LoginBean</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <property-name>nom</property-name>
    <value>test</value>
  </managed-property>
</managed-bean>
...
```

Lorsque que le bean sera instancié, JSF appellera automatiquement les setters des propriétés identifiées dans des tags <managed-property> avec les valeurs fournies dans leur tag <value> respectif.

Pour initialiser la propriété à null, il faut utiliser le tag <null-value>

Exemple :

```
...
<managed-property>
  <property-name>nom</property-name>
  <null-value>
</managed-property>
...
```

Ces informations seront utilisées par JSF pour automatiser la création ou la récupération d'un bean lorsque celui ci sera utilisé dans l'application.

Le grand intérêt de ce mécanisme est de ne pas avoir à se soucier de l'instanciation du bean ou de sa recherche dans la portée puisque c'est le framework qui va s'en occuper de façon transparente.

59.6.2. Les expressions de liaison de données d'un bean

Il est toujours nécessaire dans la partie présentation d'obtenir la valeur d'une donnée d'un bean pour par exemple l'afficher.

JSF propose une syntaxe basée sur des expressions qui facilite l'utilisation des valeurs d'un bean. Ces expressions doivent être délimitées par #{ et }.

Basiquement une expression est composée du nom du bean suivi du nom de la propriété désirée séparés par un point.

Exemple :

```
<h:inputText value="#{login.nom}"/>
```

Cet exemple affecte la valeur de l'attribut nom du bean login au composant de type saisie de texte. Dans ce cas précis, c'est aussi cet attribut de ce bean qui recevra la valeur saisie lorsque la page sera envoyée au serveur.

En fonction du contexte le résultat de l'évaluation peut conduire à l'utilisation du getter (par exemple pour afficher la valeur) ou du setter (pour affecter la valeur après un envoi de la page). C'est JSF qui le détermine en fonction du contexte.

La notation par point peut être remplacée par l'utilisation de crochets. Dans ce cas, le nom de la propriété doit être mis entre simples ou doubles quotes dans les crochets.

Exemple :

```
login.nom
login["nom"]
login['nom']
```

Ces trois expressions sont rigoureusement identiques. Cette syntaxe peut être plus pratique lors de la manipulation de collections mais elle est obligatoire lorsque la propriété contient un point.

Exemple :

```
msg["login.titre"]
```

L'utilisation des quotes simples ou doubles est équivalente car il faut les imbriquer par exemple lors de leur utilisation comme valeur de l'attribut d'un composant.

Exemple :

```
<h:inputText value='#{login["nom"]}'/>
<h:inputText value="#{login['nom']}"/>
```

Attention, la syntaxe utilisée par JSF est proche mais différente de celle proposée par JSTL : JSF utilise le délimiteur #{ ... } et JSTL utilise le délimiteur ${ ... } .

JSF définit un ensemble de variables prédéfinies, utilisables dans les expressions de liaison de données :

Variables	Rôle
header	une collection de type Map encapsulant les éléments définis dans les paramètres de l'en-tête de la requête http (seule la première valeur est renvoyée)
header-value	une collection de type Map encapsulant les éléments définis dans les paramètres de l'en-tête de la requête http (toutes les valeurs sont renvoyées sous la forme d'un tableau)
param	une collection de type Map encapsulant les éléments définis dans les paramètres de la requête http (seule la première valeur est renvoyée)
param-values	une collection de type Map encapsulant les éléments définis dans les paramètres de la requête http (toutes les valeurs sont renvoyées sous la forme d'un tableau)
cookies	une collection de type Map encapsulant les éléments définis dans les cookies
initParam	une collection de type Map encapsulant les éléments définis dans les paramètres d'initialisation de l'application
requestScope	une collection de type Map encapsulant les éléments définis dans la portée request
sessionScope	une collection de type Map encapsulant les éléments définis dans la portée session
applicationScope	une collection de type Map encapsulant les éléments définis dans la portée application
facesContext	une instance de la classe FacesContext
View	une instance de la classe UIViewRoot qui encapsule la vue

Lorsque qu'une variable est utilisée dans une expression, JSF recherche dans la liste des variables prédéfinies, puis recherche une instance dans la portée request, puis dans la portée session et enfin dans la portée application. Si aucune instance n'est trouvée, alors JSF crée une nouvelle instance en tenant compte des informations du fichier de configuration. Cette instanciation est réalisée par un objet de type VariableResolver de l'application.

La syntaxe des expressions possède aussi quelques opérateurs :

Opérateurs	Rôle	Exemple

+ - * / % div mod	opérateurs arithmétiques	
< <= > >= == != lt le gt ge eq ne	opérateurs de comparaisons	
&& || ! and or not	opérateurs logiques	<h:inputText rendered="#{!monBean.affichable}" />
Empty	opérateur vide : un objet null, une chaîne vide, un tableau ou une collection sans élément,	
? :	opérateur ternaire de test	

Il est possible de concaténer le résultat de l'évaluation de plusieurs expressions simplement en les plaçant les uns à la suite des autres.

Exemple :

```
<h:outputText value="#{messages.salutation}, #{utilisateur.nom}!"/>
```

Il est parfois nécessaire d'évaluer une expression dans le code des objets métiers pour obtenir sa valeur. Comme tous les composants sont stockés dans le FaceContext, il est possible d'accéder à cet objet pour obtenir les informations désirées. Il est d'abord nécessaire d'obtenir l'instance courante de l'objet FaceContext en utilisant la méthode statique getCurrentInstance().

Exemple :

```
FacesContext context = FacesContext.getCurrentInstance();
ValueBinding binding = context.getApplication().createValueBinding("#{login.nom}");
String nom = (String) binding.getValue(context);
```

59.6.3. Les Backing beans

Les beans de type backing bean sont spécialement utilisés avec JSF pour encapsuler tout ou partie des composants qui composent une page et ainsi faciliter leur accès notamment lors des traitements.

Ces beans sont particulièrement utiles durant des traitements réalisés lors de validations ou de traitements d'événements car ils permettent un accès aux composants dont ils possèdent une référence.

Exemple :

```
package com.jmd.test.jsf;

import javax.faces.component.UIInput;

public class LoginBean {

  private UIInput composantNom;
  private String nom;
  private String mdp;

  public UIInput getComposantNom() {
    return composantNom;
  }

  public void setComposantNom(UIInput input) {
    composantNom = input;
  }

  public String getNom() {
```

```
        return nom;
    }
...
}
```

Dans la vue, il est nécessaire de lier un composant avec son attribut correspondant dans le backing bean. L'attribut binding d'un composant permet de réaliser cette liaison.

Exemple :

```
<h:inputText value="#{login.nom}" binding="#{login.composantNom}" />
```

59.7. Les composants pour les interfaces graphiques

JSF propose un ensemble de composants serveurs pour faciliter le développement d'interfaces graphiques utilisateur.

Pour les composants, JSF propose :

- un ensemble de classes qui gèrent le comportement et l'état d'un composant
- un modèle pour assurer le rendu du composant pour un type d'application (par exemple HTML)
- un modèle de gestion des événements émis par le composant reposant sur le modèle des listeners
- la possibilité d'associer à un composant un composant de conversion de données ou de validation des données

Tous ces composants héritent de la classe abstraite UIComponentBase.

JSF propose 12 composants de base :

UICommand	Composant qui permet de réaliser une action qui lève un événement
UIForm	Composant qui regroupe d'autres composants dont l'état sera renvoyé au serveur lors de l'envoi au serveur
UIGraphic	Composant qui représente une image
UIInput	Composant qui permet de saisir des données
UIOutput	Composant qui permet d'afficher des données
UIPanel	Composant qui regroupe d'autre composant à afficher sous la forme d'un tableau
UIParameter	
UISelectItem	Composant qui représente un élément sélectionné parmi un ensemble d'éléments
UISelectItems	Composant qui représente un ensemble d'éléments
UISelectBoolean	Composant qui permet de sélectionner parmi deux états
UISelectMany	Composant qui permet de sélectionner plusieurs éléments parmi un ensemble
UISelectOne	Composant qui permet de sélectionner un seul élément parmi un ensemble

Ces classes sont des javabeans qui définissent les fonctionnalités de base des composants permettant la saisie et la sélection de données.

Chacun de ces composants possède un type, un identifiant, une ou plusieurs valeurs locales et des attributs. Ils sont extensibles et il est même possible de créer ces propres composants.

Le comportement de ces composants repose sur le traitement d'événements respectivement le modèle de gestion des événements de JSF.

Ces classes ne sont pas utilisées directement : elles sont utilisées par la bibliothèque de tags personnalisés qui se charge de les instancier et de leur associer le modèle de rendu adéquat.

Ces classes ne prennent pas en charge le rendu du composant. Par exemple, un objet de type UICommand peut être rendu en HTML sous la forme d'un lien hypertexte ou d'un bouton de formulaire.

59.7.1. Le modèle de rendu des composants

Pour chaque composant, il est possible de définir un ou plusieurs modèles qui se chargent du rendu d'un composant dans un contexte client particulier (par exemple HTML).

L'association entre un composant et son modèle de rendu est réalisée dans un RenderKit : il précise pour chaque composant quel est le ou les modèles de rendu à utiliser. Par exemple, un objet de type UISelectOne peut être rendu sous la forme d'un ensemble de bouton radio, d'une liste ou d'une liste déroulante. Chacun de ces rendus est définis par un objet de type Renderer.

L'implémentation de référence propose un seul modèle de rendu pour les composants qui propose de générer de l'HTML.

Ce modèle favorise la séparation entre l'état et le comportement d'un composant et sa représentation finale.

Le modèle de rendu permet de définir la représentation visuelle des composants. Chaque composant peut être rendu de plusieurs façons avec plusieurs modèles de rendu. Par exemple, un composant de type UICommand peut être rendu sous la forme d'un bouton ou d'un lien hypertexte. Dans cet exemple, le rendu est HTML mais il est possible d'utiliser d'autre système de rendu comme XML ou WML.

Le modèle de rendu met un oeuvre un plusieurs kits de rendus.

59.7.2. L'utilisation de JSF dans une JSP

Pour une utilisation dans une JSP, l'implémentation de référence propose deux bibliothèques de tags personnalisés :

- core : cette bibliothèque contient des fonctionnalités de bases ne générant aucun rendu. L'utilisation de cette bibliothèque est obligatoire car elle contient notamment l'élément view
- html : cette bibliothèque se charge des composants avec un rendu en HTML

Pour utiliser ces deux bibliothèques, il est nécessaire d'utiliser une directive taglib pour chacune d'elle au début de page jsp.

Exemple :

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
```

Le préfix est libre mais par convention ce sont ceux fournis dans l'exemple qui sont utilisés.

Le tag <view> est obligatoire dans toutes pages utilisant JSF. Cet élément va contenir l'état de l'arborescence des composants de la page si l'application est configurée pour stocker l'état sur le client.

Le tag <form> génère un tag HTML form qui définit un formulaire.

Exemple :

```
<html>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<f:view>
<head>
  <title>Application de tests avec JSF</title>
```

```
</head>
<body>
  <h:form>
  ...
  </h:form>
</body>
</f:view>
</html>
```

59.8. La bibliothèque de tags Core

Cette bibliothèque est composée de 18 tags.

Tag	Rôle
actionListener	ajouter un listener pour une action sur composant
attribute	ajouter un attribut à un composant
convertDateTime	ajouter un convertisseur de type DateTime à un composant
convertNumber	ajouter un convertisseur de type numérique à un composant
facet	définit un élément particulier d'un composant
loadBundle	charger un fichier contenant les chaînes de caractères d'une locale dans une collection de type Map
param	ajouter un paramètre à un composant
selectitem	définir l'élément sélectionné dans un composant permettant de faire un choix
selectitems	définir les éléments sélectionnés dans un composant permettant de faire un choix
subview	définir une sous vue
verbatim	ajouter un texte brut à la vue
view	définir une vue
validator	ajouter un valideur à un composant
validateDoubleRange	ajouter un valideur de type « plage de valeurs réelles » à un composant
validateLength	ajouter un valideur de type « taille de la valeur » à un composant
validateLongRange	ajouter un valideur de type « plage de valeurs entières » à un composant
valueChangeListener	ajouter un listener pour un changement de valeur sur un composant

La plupart de ces tags permettent d'ajouter des objets à un composant. Leur utilisation sera détaillée tout au long de ce chapitre.

59.8.1. Le tag <selectItem>

Ce tag représente un élément dans un composant qui peut en contenir plusieurs.

Les attributs de base sont les suivants :

Attribut	Rôle
itemValue	contient la valeur de l'élément

itemLabel	contient le libellé de l'élément
itemDescription	contient une description de l'élément (utilisé uniquement par les outils de développement)
itemDisabled	contient l'état de l'élément
binding	contient le nom d'une méthode qui renvoie un objet de type javax.faces.model.SelectItem
id	contient l'identifiant du composant
value	contient une expression qui désigne un objet de type javax.faces.model.SelectItem

Exemple :

```
<f:selectItem value="#{test.elementSelectionne}"/>
```

L'attribut value attend en paramètre une expression qui désigne une méthode qui renvoie un objet de type SelectItem qui encapsule l'objet de la liste qui sera sélectionné.

Exemple :

```
...
public SelectItem getElementSelectionne() {
  return new SelectItem("Element 1");
}
...
```

La classe SelectItem possède quatre constructeurs qui permettent de définir les différentes propriétés qui composent l'élément.

59.8.2. Le tag <selectItems>

Ce tag représente une collection d'éléments dans un composant qui peut en contenir plusieurs.

Ce tag est particulièrement utile car il évite d'utiliser autant de tag selectItem que d'éléments à définir.

Exemple :

```
...
<h:selectOneRadio>
  <f:selectItems value="#{test.listeElements}"/>
</h:selectOneRadio>
...
```

La collection d'objets de type SelectItem peut être soit une collection soit un tableau.

Exemple : avec un tableau d'objects de type SelectItem

```
package com.jmd.test.jsf;

import javax.faces.model.SelectItem;

public class TestBean {

  private SelectItem[] elements = {
    new SelectItem(new Integer(1), "Element 1"),
    new SelectItem(new Integer(2), "Element 2"),
    new SelectItem(new Integer(3), "Element 3"),
    new SelectItem(new Integer(4), "Element 4"),
  };

  public SelectItem[] getListeElements() {
```

```
    return elements;
  }

...
}
```

La collection peut être de type Map : dans ce cas le framework associe la clé de chaque occurrence à la propriété itemValue et la valeur à la propriété itemLabel

Exemple :

```
package com.jmd.test.jsf;

import java.util.HashMap;
import java.util.Map;

import javax.faces.model.SelectItem;

public class TestBean {

  private Map elements = null;

  public Map getListeElements() {
    if (elements == null) {
      elements = new HashMap();
      elements.put("Element 1", new Integer(1));
      elements.put("Element 2", new Integer(2));
      elements.put("Element 3", new Integer(3));
      elements.put("Element 4", new Integer(4));
    }
    return elements;
  }

  public SelectItem getElementSelectionne() {
    return new SelectItem("Element 1");
  }

  ...
}
```

59.8.3. Le tag <verbatim>

Ce tag permet d'insérer du texte dans la vue.

Son utilisation est obligatoire dans le corps des tags JSF pour insérer autre chose qu'un tag JSF. Par exemple, pour insérer un tag HTML dans le corps d'un tag JSF, il est obligatoire d'utiliser le tag <verbatim>.

Les tags suivants peuvent avoir un corps : commandLink, outputLink, panelGroup, panelGrid et dataTable.

Exemple :

```
<h:outputLink value="http://java.sun.com" title="Java">
  <f:verbatim>
    Site Java de Sun
  </f:verbatim>
</h:outputLink>
```

Il est possible d'utiliser le tag <outputText> à la place du tag <verbatim>.

59.8.4. Le tag <attribute>

Ce tag permet de fournir un attribut quelconque à un composant puisque chaque composant peut stocker des attributs arbitraires.

Ce tag possède deux attributs :

Attribut	Rôle
Name	nom de l'attribut
Value	valeur de l'attribut

Dans le code d'un composant, il est possible d'utiliser la méthode getAttributes() pour obtenir une collection de type Map des attributs du composant.

Ceci permet de fournir un mécanisme souple pour fournir des paramètres sans être obligé de créer un nouveau composant ou de modifier un composant existant en lui ajoutant un ou plusieurs attributs.

59.8.5. Le tag <facet>

Ce tag permet de définir des éléments particuliers d'un composant.

Il est par exemple utilisé pour définir les lignes d'en-tête et de pied de page des tableaux.

Ce tag possède plusieurs attributs :

Attribut	Rôle
Name	Permet de préciser le type de l'élément généré par le tag Les valeurs possibles sont header et footer

Exemple :

```
<h:column>
  <f:facet name="header">
    <h:outputText value="Nom" />
  </f:facet>
  <h:outputText value="#{personne.nom}"/>
</h:column>
```

59.9. La bibliothèque de tags Html

Cette bibliothèque est composée de 25 tags qui permettent la réalisation de l'interface graphique de l'application.

Tag	Rôle
form	le tag <form> HTML
commandButton	un bouton
commandLink	un lien qui agit comme un bouton
graphicImage	une image
inputHidden	une valeur non affichée
inputSecret	une zone de saisie de texte mono ligne dont la valeur est non lisible

inputText	une zone de saisie de texte mono ligne
inputTextarea	une zone de saisie de texte multi-lignes
outputLink	un lien
outputFormat	du texte affiché avec des valeurs fournies en paramètre
outputText	du texte affiché
panelGrid	un tableau
panelGroup	un panneau permettant de regrouper plusieurs composants
selectBooleanCheckbox	une case à cocher
selectManyCheckbox	un ensemble de cases à cocher
selectManyListbox	une liste déroulante où plusieurs éléments sont sélectionnables
selectManyMenu	un menu où plusieurs éléments sont sélectionnables
selectOneListbox	une liste déroulante où un seul élément est sélectionnable
selectOneMenu	un menu où un seul élément est sélectionnable
selectOneRadio	un ensemble de boutons radio
dataTable	une grille proposant des fonctionnalités avancées
column	une colonne d'une grille
message	le message d'erreur lié à un composant
messages	les messages d'erreur liés à tous les composants

59.9.1. Les attributs communs

Ces tags possèdent des attributs communs pouvant être regroupés en trois catégories :

- les attributs de base
- les attributs liés à HTML
- les attributs liés à Javascript

Chaque tag utilise ou non chacun de ces attributs.

Les attributs de base sont les suivants :

Attribut	Rôle
id	contient l'identifiant du composant
binding	permet l'association avec un backing bean
rendered	contient un booléen qui indique si le composant doit être affiché
styleClass	contient le nom d'une classe CSS à appliquer au composant
value	contient la valeur du composant
valueChangeListener	permet l'association à une méthode qui va traiter les changements de valeurs
converter	contient une classe de conversion des données de chaîne de caractères en objet et vice et versa
validator	contient une classe de validation des données
required	contient un booléen qui indique si une valeur doit obligatoirement être saisie

L'attribut id est très important car il permet d'avoir accès :

- au tag dans le code de la vue par d'autres tags
  <h:inputText id="nom" required="true"/>

<h:message for="nom"/>

- au tag dans le code Javascript de la vue
- dans le code Java des objets métiers.
  UIComponent component = event.getComponent().findComponent("nomComposant");

L'attribut binding permet d'associer le composant avec un champ d'une classe de type bean. Un tel bean est nommé backing bean dans une application JSF.

Exemple :

```
...
<h:inputText value="#{login.nom}" id="nom" required="true" binding="#{login.inputTextNom}"/>
..

...
import javax.faces.component.UIComponent;

public class LoginBean {
  private String nom;

  private UIComponent inputTextNom;

  public UIComponent getInputTextNom() {
    return inputTextNom;
  }

  public void setInputTextNom(UIComponent inputTextNom) {
    this.inputTextNom = inputTextNom;
  }
...
```

L'attribut value permet de préciser la valeur d'un tag. Cette valeur peut être fournie sous deux formes :

- en dur dans le code :
  <h:outputText value="Bonjour"/>
- en utilisant une expression de liaison de données :
  <h:inputText value="#{login.nom}"/>

L'attribut converter permet de préciser une classe qui va convertir la valeur d'un objet en chaîne de caractères et vice et versa. L'utilisation de cet attribut est détaillée dans une des sections suivante.

L'attribut validator permet de préciser une classe qui va réaliser des contrôles de validation sur la valeur saisie. L'utilisation de cet attribut est détaillée dans une des sections suivante.

L'attribut styleClass permet de préciser le nom d'un style défini dans une feuille de style CSS qui sera appliqué au composant.

Exemple : le fichier monstyle.css

```
.titre {
color:red;
}
```

Dans la vue, il faut inclure la feuille de style dans la partie en-tête de la page HTML.

Exemple :

```
...
<link href="monstyle.css" rel="stylesheet" type="text/css"/>
...
<h:outputText value="#{msg.login_titre}" styleClass="titre"/>
...
```

L'attribut renderer permet de préciser si le composant sera affiché ou non dans la vue. La valeur de l'attribut peut être obtenue dynamiquement par l'utilisation du langage d'expression.

Exemple :

```
<h:panelGrid rendered='#{listepersonnes.nbOccurrences gt 0}'/>
```

Les principaux attributs liés à HTML sont les suivants :

Attribut	Rôle
accesskey	contient le raccourci clavier pour donner le focus au composant
alt	contient le texte alternatif pour les composants non textuels
border	contient la taille de la bordure en pixel
disabled	permet de désactiver le composant
maxlength	contient le nombre maximum de caractères saisis
readonly	permet de rendre une zone de saisie en lecture seule
rows	contient nombre de lignes visibles pour zone de saisie multi-ligne
shape	contient la définition d'une région
size	contient la taille de la zone de saisie
style	contient le style CSS à utiliser
target	contient le nom de la frame cible pour l'affichage de la page
title	contient le titre du composant généralement transformé en une bulle d'aide
width	contient la taille du composant

Le rôle de la plupart de ces tags est identique à leurs homologues définis dans HTML 4.0.

L'attribut style permet de définir un style CSS qui sera appliqué au composant. Cet attribut contient directement la définition du style à la différence de l'attribut styleClass qui contient le nom d'une classe CSS définit dans une feuille de style. Il est préférable d'utiliser l'attribut styleClass plutôt que l'attribut style afin de faciliter la maintenance de la charte graphique.

Exemple :

```
<h:outputText value="#{login.nom}" style="color:red;"/>
```

Les attributs liés à Javascript sont :

Attribut	Rôle
onblur	perte du focus
onchange	changement de la valeur
onclick	clic du bouton de la souris sur le composant

ondblclick	double-clic du bouton de la souris sur le composant
onfocus	réception du focus
onkeydown	une touche est enfoncée
onkeypress	appui sur une touche
onkeyup	une touche est relachée
onmousedown	
onmousemove	déplacement du curseur de la souris sur le composant
onmouseout	déplacement hors du cuseur de la souris hors du composant
onmouseover	passage de la souris au dessus du composant
onmouseup	le bouton de la souris est relachée
onreset	réinitialisation du formulaire
onselect	sélection du texte dans une zone de saisie
onsubmit	soumission du formulaire

59.9.2. Le tag <form>

Ce tag représente un formulaire HTML.

Il possède les attributs suivants :

Attributes	Rôle
binding, id, rendered, styleClass	attributs communs de base
accept, acceptcharset, dir, enctype, lang, style, target, title	attributs communs liés à HTML
onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onreset, onsubmit	attributs communs liés aux événements Javascript

Il est préférable de définir explicitement l'attribut id pour permettre son exploitation notamment dans le code Javascript, sinon un id est généré automatiquement.

Ceci est d'autant plus important que les id des composants intégrés dans le formulaire sont préfix par l'id du formulaire suivi du caractère deux points. Il faut tenir compte de ce point lors de l'utilisation de code Javascript faisant référence à un composant.

59.9.3. Les tags <inputText>, <inputTextarea>, <inputSecret>

Ces trois composants permettent de générer des composants pour la saisie de données.

Les attributs de ces tags sont les suivants :

Attributs	Rôle
cols	

	définir le nombre de colonne (pour le composant inputTextarea uniquement)
immediate	permettre de demander d'ignorer les étapes de validation des données
redisplay	permettre de réafficher le contenu lors du réaffichage de la page (pour le composant inputSecret uniquement)
required	rendre obligatoire la saisie d'une valeur
rows	définir le nombre de lignes affichées (pour le composant inputTextarea uniquement)
valueChangeListener	préciser une classe de type listener lors du changement de la valeur
binding, converter, id, rendered, required, styleClass, value, validator	attributs communs de base
accesskey, alt, dir, disabled,lang, maxlength, readonly, size, style, tabindex, title	attributs communs liés à HTML
onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onselect	attributs communs lies aux événements Javascript

Exemple :

```
<html>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<f:view>
<head>
  <title>Saisie des données</title>
</head>
<body>
  <h:form>
    <h3>Saisie des données</h3>
    <p><h:inputText size="20" /></p>
    <p><h:inputTextarea rows="3" cols="20" /></p>
    <p><h:inputSecret size="20" /></p>
  </h:form>
</body>
</f:view>
</html>
```

Résultat

texte

ligne1
ligne2
ligne3

******

59.9.4. Le tag <ouputText> et <outputFormat>

Ces deux tags permettent d'insérer une valeur sous la forme d'une chaîne de caractères dans la vue. Par défaut, ils ne

génèrent pas de tag HTML mais insèrent simplement la valeur dans la vue sauf si un style CSS est précisé avec l'attribut style ou styleClass. Dans ce cas, la valeur est contenue un tag HTML <span>.

Les attributs de ces deux tags sont :

Attributs	Rôle
escape	booléen qui précise si certains caractères de la valeur seront encodé ou non. La valeur par défaut est false.
binding, converter, id, rendered, styleClass, value	attributs communs de base
style, title	attributs communs lies à HTML

L'attribut escape est particulièrement utile pour encoder certains caractères spéciaux avec leur code HTML correspondant.

Exemple :

```
<h:outputText escape="true" value="Nombre d'occurrences > 200"/>
```

Le tag outputText peut être utilisé pour générer du code HTML en valorisant l'attribut escape à false.

Exemple :

```
<p><h:outputText escape="false" value="<H2>Saisie des données</H2>"/></p>

<p><h:outputText escape="true" value="<H2>Saisie des données</H2>"/></p>
```

Résultat :

Saisie des données

<H2>Saisie des données</H2>

Le tag outputFormat permet de formater une chaîne de caractères avec des valeurs fournies en paramètres.

Exemple :

```
<p>
  <h:outputFormat value="La valeur doit être entre {0} et {1}.">
    <f:param value="1"/>
    <f:param value="9"/>
  </h:outputFormat>
</p>
```

Résultat

La valeur doit être entre 1 et 9.

Ce composant utilise la classe java.text.MessageFormat pour formater le message. L'attribut value doit donc contenir une chaîne de caractères utilisable par cette classe.

Le tag <param> permet de fournir la valeur de chacun des paramètres.

59.9.5. Le tag <graphicImage>

Ce composant représente une image : il génère un tag HTML <img>.

Les attributs de ce tag sont les suivants :

Attributs	Rôle
binding, id, rendered, styleClass, value	Attributs communs de base
alt, dir, height, ismap, lang, longdesc, style, title, url, usemap, width	Attributs communs lies à HTML
onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup	Attributs communs lies aux événements Javascript

Les attributs value et url peuvent préciser l'url de l'image.

Exemple :

```
<p><h:graphicImage value="/images/erreur.jpg" /></p>
<p><h:graphicImage url="/images/warning.jpg" /></p>
```

Résultat

59.9.6. Le tag <inputHidden>

Ce composant représente un champ caché dans un formulaire.

Les attributs sont les suivants :

Attributs	Rôle
binding, converter, id, immediate, required, validator, value, valueChangeListener	attributs communs de base

Exemple :

```
<h:inputHidden value="#{login.nom}" />
```

Résultat dans le code HTML:

```
...
          <input type="hidden" name="_id0:_id12" value="test" />
...
```

59.9.7. Le tag <commandButton> et <commandLink>

Ces composants représentent respectivement un bouton de formulaire et un lien qui déclenche une action. L'action demandée sera traitée par le framework JSF.

Les attributs sont les suivants :

Attributs	Rôle
action	peut être une chaîne de caractères ou une méthode qui renvoie une chaîne de caractères qui sera traitée par le navigation handler.
actionListener	précise une méthode possédant une signature void nomMethode(ActionEvent) qui sera exécutée lors d'un clic
image	url, tenant compte du contexte de l'application, de l'image qui sera utilisée à la place du bouton (uniquement pour le tag commandButton)
type	type de bouton généré : button, submit, reset (uniquement pour le tag commandButton)
value	le texte affiché par le bouton ou le lien
accesskey, alt, binding, id, lang, rendered, styleClass	attributs communs de base
coords (uniquement pour le tag commandLink), dir, disabled, hreflang (uniquement pour le tag commandLink), lang, readonly, rel (uniquement pour le tag commandLink), rev (uniquement pour le tag commandLink), shape (uniquement pour le tag commandLink), style, tabindex, target (uniquement pour le tag commandLink), title	attributs communs lies à HTML
onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup, onselect	attributs communs lies aux événements Javascript

Il est possible d'insérer dans le corps du tag <commandLink> d'autres composants qui feront partie intégrante du lien comme par exemple du texte ou une image.

Exemple :

```
<p>
  <h:commandLink>
    <h:outputText value="Valider"/>
  </h:commandLink>
</p>
<p>
  <h:commandLink>
    <h:graphicImage value="/images/oeil.jpg"/>
  </h:commandLink>
</p>
```

Résultat

Valider

Il est aussi possible de fournir un ou plusieurs paramètres qui seront envoyés dans la requête en utilisant le tag <param> dans le corps du tag

Exemple :

```
<h:commandLink>
  <h:outputText value="Selectionner"/>
  <f:param name="id" value="1"/>
</h:commandLink>
```

Résultat dans le page HTML générée :

```
<a href="#" onclick="document.forms['_id0']['_id0:_idcl'].value='_id0:_id15';
          document.forms['_id0'].submit(); return false;">
        mg src="/test_JSF/images/oeil.jpg" alt="" /></a>
```

Le tag <commandLink> génère dans la vue du code Javascript pour soumettre le formulaire lors d'un clic.

59.9.8. Le tag <ouputLink>

Ce composant représente un lien direct vers une ressource dont la demande ne sera pas traitée par le framework JSF.

Les attributs sont les suivants :

Attributs	Rôle
accesskey, binding, converter, id, lang, rendered, styleClass, value	attributs communs de base
charset, coords, dir, hreflang, lang, rel, rev, shape, style, tabindex, target, title, type	attributs communs lies à HTML
onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup	attributs communs lies aux événements Javascript

L'attribut value doit contenir l'url qui sera utilisée dans l'attribut href du lien HTML. Si le première caractère est un # (dièse) alors le lien pointe vers une ancre définie dans la même page.

Il est possible d'insérer dans le corps du tag ouputLink d'autres composants qui feront partie intégrante du lien.

Exemple :

```
<p>
  <h:outputLink value="http://java.sun.com">
    <h:graphicImage value="/images/java.jpg"/>
  </h:outputLink>
</p>
```

Résultat

Le code HTML généré dans la page est le suivant :

Attention, pour mettre du texte dans le corps du tag, il est nécessaire d'utiliser un tag verbatim ou outputText.

Exemple :

```
<p>
  <h:outputLink value="http://java.sun.com" title="Java">
    <f:verbatim>
       Site Java de Sun
    </f:verbatim>
  </h:outputLink>
</p>
```

59.9.9. Les tags <selectBooleanCheckbox> et <selectManyCheckbox>

Ces composants représentent respectivement une case à cocher et un ensemble de cases à cocher.

Les attributs sont les suivants :

Attributs	Rôle
disabledClass	classe CSS pour les éléments non sélectionnés (pour le tag selectManyCheckbox uniquement)
enabledClass	classe CSS pour les éléments sélectionnés (pour le tag selectManyCheckbox uniquement)
layout	préciser la disposition des éléments (pour le tag selectManyCheckbox uniquement)
binding, converter, id, immediate, styleClass, required, rendered, validator, value, valueChangeListener	attributs communs de base
accesskey, border, dir, disabled, lang, readonly, style, tabindex, title	attributs communs lies à HTML (border pour le tag selectManyCheckbox uniquement)
onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup, onselect	attributs communs lies aux événements Javascript

L'attribut layout permet de préciser la disposition des cases à cocher : lineDirection pour une disposition horizontale (c'est la valeur par défaut) et pageDirection pour une disposition verticale.

Le tag <selectBooleanCheckbox> dont la valeur peut être associée à une propriété booléenne d'un bean représente une case à cocher simple.

Exemple :

```
<h:selectBooleanCheckbox value="#{saisieOptions.recevoirLettre}">
</h:selectBooleanCheckbox> Recevoir la lettre d'information
```

Résultat :

Pour gérer l'état du composant, il faut utiliser l'attribut value en lui fournissant en valeur une propriété booléen d'un backing bean.

Exemple :

```
public class SaisieOptions {

  private boolean recevoirLettre;

  public void setRecevoirLettre(boolean valeur) {
    recevoirLettre = valeur;
  }

  public boolean getRecevoirLettre() {
    return recevoirLettre;
  }
...
```

Le tag <selectManyCheckbox> représente un ensemble de cases à cocher. Dans cet ensemble, il est possible de sélectionner une ou plusieurs cases à cocher.

Chaque case à cocher est définie par un tag selectItem dans le corps du tag selectMenyCheckbox.

Exemple :

```
<h:selectManyCheckbox layout="pageDirection">
  <f:selectItem itemValue="petit" itemLabel="Petit" />
  <f:selectItem itemValue="moyen" itemLabel="Moyen" />
  <f:selectItem itemValue="grand" itemLabel="Grand" />
  <f:selectItem itemValue="tresgrand" itemLabel="Tres grand" />
</h:selectManyCheckbox>
```

Résultat :

☐ Petit
☐ Moyen
☐ Grand
☐ Tres grand

Le rendu du composant est un tableau HTML dont chaque cellule contient une case à cocher encapsulée dans un tag HTML <label> :

Exemple :

```
<table>
 <tr>
  <td>
   <label><input name="_id0:_id1" value="petit" type="checkbox"> Petit</input></label></td>
 </tr>
 <tr>
  <td>
   <label><input name="_id0:_id1" value="moyen" type="checkbox"> Moyen</input></label></td>
 </tr>
 <tr>
  <td>
   <label><input name="_id0:_id1" value="grand" type="checkbox"> Grand</input></label></td>
 </tr>
 <tr>
  <td>
    <label><input name="_id0:_id1" value="tresgrand" type="checkbox"> Tres grand</input>
    </label></td>
 </tr>
</table>
```

59.9.10. Le tag <selectOneRadio>

Ce composant représente un ensemble de boutons radio dont un seul peut être sélectionné.

Les attributs sont les suivants :

Attributs	Rôle
disabledClass	classe CSS pour les éléments non sélectionnés
enabledClass	classe CSS pour les éléments sélectionnés
layout	préciser la disposition des éléments
binding, converter, id, immediate, styleClass, required, rendered, validator, value, valueChangeListener	Attributs communs de base
accesskey, border, dir, disabled, lang, readonly, style, tabindex, title	Attributs communs lies à HTML
onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup, onselect	Attributs communs lies aux événements Javascript

Les éléments peuvent être précisés un par un avec le tag <selectItem>.

Exemple :

```
<h:selectOneRadio layout="pageDirection">
  <f:selectItem itemValue="petit" itemLabel="Petit" />
  <f:selectItem itemValue="moyen" itemLabel="Moyen" />
  <f:selectItem itemValue="grand" itemLabel="Grand" />
  <f:selectItem itemValue="tresgrand" itemLabel="Tres grand" />
</h:selectOneRadio>
```

Résultat :

Petit
Moyen
Grand
Tres grand

Le rendu du composant est un tableau HTML dont chaque cellule contient un bouton radio encapsulé dans un tag HTML <label> :

Exemple :

```
<table>
  <tr>
    <td>
      <label><input type="radio" name="_id0:_id1" value="petit"> Petit</input></label></td>
  </tr>
  <tr>
    <td>
      <label><input type="radio" name="_id0:_id1" value="moyen"> Moyen</input></label></td>
  </tr>
```

```
  <tr>
    <td>
      <label><input type="radio" name="_id0:_id1" value="grand"> Grand</input></label></td>
  </tr>
  <tr>
    <td>
      <label><input type="radio" name="_id0:_id1" value="tresgrand"> Tres grand</input>
          </label></td>
  </tr>
</table>
```

Les éléments peuvent être précisés sous la forme d'un tableau de type SelecItem avec le tag <selectItems>.

Exemple :

```
<h:selectOneRadio value="#{saisieOptions.taille}" layout="pageDirection" id="taille">
<f:selectItems value="#{saisieOptions.tailleItems}"/>
</h:selectOneRadio>
```

Dans ce cas, le bean doit contenir au moins deux méthodes : getTaille() pour renvoyer la valeur de l'élément sélectionner et getTailleItems() qui renvoie un tableau d'objets de type SelectItems contenant les éléments.

Exemple :

```
package com.jmd.test.jsf;

import javax.faces.model.*;

public class SaisieOptions {

  private Integer taille = null;

  private SelectItem[] tailleItems = {
    new SelectItem(new Integer(1), "Petit"),
    new SelectItem(new Integer(2), "Moyen"),
    new SelectItem(new Integer(3), "Grand"),
    new SelectItem(new Integer(4), "Très grand") };

  public SaisieOptions() {
    taille = new Integer(2);
  }

  public Integer getTaille() {
    return taille;
  }

  public void setTaille(Integer newValue) {
    taille = newValue;
  }

  public SelectItem[] getTailleItems() {
    return tailleItems;
  }
}
```

Le bean doit être déclaré dans le fichier faces-config.xml

Exemple :

```
<managed-bean>
  <managed-bean-name>saisieOptions</managed-bean-name>
  <managed-bean-class>com.jmd.test.jsf.SaisieOptions</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

Résultat :

Petit
Moyen
Grand
Très grand

59.9.11. Le tag <selectOneListbox>

Ce composant représente une liste d'éléments dont un seul peut être sélectionné

Les attributs sont les suivants :

Attributs	Rôle
binding, converter, id, immediate, styleClass, required, rendered, validator, value, valueChangeListener	attributs communs de base
accesskey, dir, disabled, lang, readonly, style, size, tabindex, title	attributs communs lies à HTML
onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup, onselect	attributs communs lies aux événements Javascript

L'attribut size permet de préciser le nombre d'éléments de la liste affiché.

Exemple :

```
<h:selectOneListbox value="#{saisieOptions.taille}">
  <f:selectItems value="#{saisieOptions.tailleItems}"/>
</h:selectOneListbox>
```

Résultat :

59.9.12. Le tag <selectManyListbox>

Ce composant représente une liste d'éléments dont plusieurs peuvent être sélectionnés.

Les attributs sont les suivants :

Attributs	Rôle
binding, converter, id, immediate, styleClass, required, rendered, validator, value, valueChangeListener	attributs communs de base
accesskey, dir, disabled, lang, readonly, style, size, tabindex, title	attributs communs lies à HTML

onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup, onselect	attributs communs lies aux événements Javascript

Exemple :

```
<h:selectManyListbox value="#{saisieOptions.legumes}">
  <f:selectItems value="#{saisieOptions.legumesItems}"/>
</h:selectManyListbox>
```

La liste des éléments sélectionnés doit pouvoir contenir zéro ou plusieurs valeurs sous la forme d'un tableau ou d'une liste.

Exemple :

```
package com.jmd.test.jsf;

import javax.faces.model.*;

public class SaisieOptions {

  private String[] legumes = {
    "navets", "choux" };
  private SelectItem[] legumesItems = {
    new SelectItem("epinards", "Epinards"),
    new SelectItem("poireaux", "Poireaux"),
    new SelectItem("navets", "Navets"),
    new SelectItem("flageolets", "Flageolets"),
    new SelectItem("choux", "Choux"),
    new SelectItem("aubergines", "Aubergines") };

  public SaisieOptions() {
  }

  public String[] getLegumes() {
    return legumes;
  }

  public SelectItem[] getLegumesItems() {
    return legumesItems;
  }
}
```

Résultat :

Il est possible d'utiliser un objet de type List à la place des tableaux.

Exemple :

```
package com.jmd.test.jsf;

import javax.faces.model.*;
import java.util.*;

public class SaisieOptions {

  private List legumes = null;

  private List legumesItems = null;
```

```
  public List getLegumesItems() {
    if (legumesItems == null) {
      legumesItems = new ArrayList();
      legumesItems.add(new SelectItem("epinards", "Epinards"));
      legumesItems.add(new SelectItem("poireaux", "Poireaux"));
      legumesItems.add(new SelectItem("navets", "Navets"));
      legumesItems.add(new SelectItem("flageolets", "Flageolets"));
      legumesItems.add(new SelectItem("choux", "Choux"));
      legumesItems.add(new SelectItem("aubergines", "Aubergines"));
    }
    return legumesItems;
  }

  public List getLegumes() {
    return legumes;
  }

  public void setLegumes(List newValue) {
    legumes = newValue;
  }

  public SaisieOptions() {
    legumes = new ArrayList();
    legumes.add("navets");
    legumes.add("choux");
  }
}
```

59.9.13. Le tag <selectOneMenu>

Ce composant représente une liste déroulante dont un seul élément peut être sélectionné.

Les attributs sont les suivants :

Attributs	Rôle
binding, converter, id, immediate, styleClass, required, rendered, validator, value, valueChangeListener	attributs communs de base
accesskey, dir, disabled, lang, readonly, style, tabindex, title	attributs communs lies à HTML
onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup, onselect	attributs communs lies aux événements Javascript

Exemple :

```
<h:selectOneMenu value="#{saisieOptions.taille}">
<f:selectItems value="#{saisieOptions.tailleItems}"/>
</h:selectOneMenu>
```

Résultat :

Epinards

Exemple : le code HTML généré

```
<select name="_id0:_id6" size="1"> <option value="1">Petit</option>
  <option value="2" selected="selected">Moyen</option>
  <option value="3">Grand</option>
  <option value="4">Tr&egrave;s grand</option>
</select>
```

59.9.14. Le tag <selectManyMenu>

Ce composant représente une liste d'éléments dont le rendu HTML est un tag select avec une seule option visible.

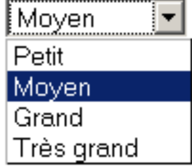
Les attributs sont les suivants :

Attributs	Rôle
binding, converter, id, immediate, styleClass, required, rendered, validator, value, valueChangeListener	Attributs communs de base
accesskey, dir, disabled, lang, readonly, style, tabindex, title	Attributs communs lies à HTML
onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup, onselect	Attributs communs lies aux événements Javascript

Exemple :

```
<h:selectManyMenu value="#{saisieOptions.legumes}">
<f:selectItems value="#{saisieOptions.legumesItems}"/>
</h:selectManyMenu>
```

Résultat :



59.9.15. Les tags <message> et <messages>

Des messages peuvent être émis lors de traitements. Ils sont stockés dans le contexte de l'application JSF pour être restitués dans la vue. Ils permettent notamment de fournir des messages d'erreur aux utilisateurs.

JSF défini quatre types de message :

- Information
- Warning
- Error
- Fatal

Chaque message possède un résumé et un descriptif.

Le tag <messages> permet d'afficher tous les messages stockés dans le contexte de l'application JSF.

Le tag message permet d'afficher un seul message, le dernier ajouté, pour un composant donné.

Les attributs sont les suivants :

Attributs	Rôle
errorClass	nom d'une classe CSS pour un message de type error

errorStyle	style CSS pour un message de type error
fatalClass	nom d'une classe CSS pour un message de type fatal
fatalStyle	style CSS pour un message de type fatal
globalOnly	booléen qui permet de n'affiche que les messages qui ne sont pas associés à un composant. Par défaut, False (uniquement pour le tag messages)
infoClass	nom d'une classe CSS pour un message de type Information
infoStyle	style CSS pour un message de type Information
Layout	format de la liste de messages : list ou table (uniquement pour le composant messages)
showDetail	booléen qui précise si la description des messages est affichée ou non. Par défaut, false pour le tag message et true pour le tag messages
showSummary	booléen qui précise si le résumé des messages est affichée ou non. Par défaut, true pour le tag message et false pour le tag messages
Tooltip	booléen qui précise si la description est afficher sous la forme d'une bulle d'aide
warnClass	nom d'une classe CSS pour un message de type Warning
warnStyle	Style CSS pour un message de type Warning
For	l'identifiant du composant pour lequel le message doit être affiché
binding, id, rendered, styleClass	attributs communs de base
style, title	attributs communs lies à HTML

59.9.16. Le tag <panelGroup>

Ce composant permet de regrouper plusieurs composants.

Les attributs sont les suivants :

Attributs	Rôle
binding, id, rendered, styleClass	attributs communs de base
Style	attributs communs lies à HTML

Exemple :

```
<td bgcolor='#DDDDDD'>
  <h:panelGroup>
    <h:inputText value="#{login.nom}" id="nom" required="true"/>
    <h:message for="nom"/>
  </h:panelGroup>
</td>
```

Résultat :

Erreur de validation: Valeur requise.

59.9.17. Le tag <panelGrid>

Ce composant représente un tableau HTML.

Les attributs sont les suivants :

Attributs	Rôle
Bgcolor	couleur de fond du tableau
Border	taille de la bordure du tableau
Cellpadding	espacement intérieur de chaque cellule
Cellspacing	espacement extérieur de chaque cellule
columnClasses	nom de classes CSS pour les colonnes. Il est possible de préciser plusieurs noms de classe qui seront utilisées sur chaque colonne
Columns	nombre de colonne du tableau
footerClass	nom de la classe CSS pour le pied du tableau
Frame	précise les règles pour le contour du tableau. Les valeurs possibles sont : none, above, below, hsides, vsides, lhs, rhs, box, border
headerClass	nom de la classe CSS pour l'en-tête du tableau
rowClasses	nom de classes CSS pour les lignes. Il est possible de préciser deux noms de classe séparés par une virgule qui seront utilisés alternativement sur chaque ligne
Rules	précise les règles de dessin des lignes entre les cellules. Les valeurs possibles sont : groups, rows, columns, all
Summary	résumé du tableau
binding, id, rendered, styleClass, value	attributs communs de base
dir, lang, style, title, width	attributs communs lies à HTML
onclick, ondblclick, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup	attributs communs lies aux événements Javascript

Par défaut chaque composant est inséré les uns à la suite des autres dans les cellules en partant de la gauche vers la droite en passant à ligne suivante dès que nécessaire.

Il n'est possible de mettre qu'un seul composant par cellule. Ainsi pour mettre plusieurs composants dans une cellule, il faut les regrouper dans un tag panelGroup.

Exemple :

```
<h:panelGrid columns="2">
  <h:outputText value="Nom : " />
  <h:panelGroup>
    <h:inputText value="#{login.nom}" id="nom" required="true"/>
    <h:message for="nom"/>
  </h:panelGroup>
  <h:outputText value="Mot de passe :" />
  <h:inputSecret value="#{login.mdp}"/>
  <h:commandButton value="Login" action="login"/>
</h:panelGrid>
```

Résultat :

Nom :
Mot de passe :
Login

Le code HTML généré est le suivant :

Exemple : le code HTML généré

```
...
<table>

  <tbody>
    <tr>
      <td>Nom : </td>
      <td><input id="_id0:nom" type="text" name="_id0:nom" /></td>
    </tr>
    <tr>
      <td>Mot de passe :</td>
      <td><input type="password" name="_id0:_id6" value="" /></td>
    </tr>
    <tr>
      <td><input type="submit" name="_id0:_id7" value="Login" /></td>
    </tr>
  </tbody>
</table>
...
```

59.9.18. Le tag <dataTable>

Ce composant représente un tableau HTML dans lequel des données vont pouvoir être automatiquement présentées. Ce composant est sûrement le plus riche en fonctionnalité et donc le plus complexe des composants fournis en standard.

Les attributs sont les suivants :

Attributs	Rôle
Bgcolor	couleur de fond du tableau
Border	taille de la bordure du tableau
Cellpadding	espacement intérieur de chaque cellule
Cellspacing	espacement extérieur de chaque cellule
columnClasses	nom de classes CSS pour les colonnes. Il est possible de préciser plusieurs noms de classe qui seront utilisées sur chaque colonne
First	index de la première occurrences des données qui sera affichée dans le tableau
footerClass	nom de la classe CSS pour le pied du tableau
Frame	précise les règles pour le contour du tableau. Les valeurs possibles sont : none, above, below, hsides, vsides, lhs, rhs, box, border
headerClass	nom de la classe CSS pour l'en-tête du tableau
rowClasses	nom de classes CSS pour les lignes. Il est possible de préciser deux noms de classe qui seront utilisées alternativement sur chaque ligne

Rules	précise les règles de dessin des lignes entre les cellules. Les valeurs possibles sont : groups, rows, columns, all
Summary	résumé du tableau
Var	nom de la variable qui va contenir l'occurrence en cours de traitement lors du parcours des données
binding, id, rendered, styleClass, value	attributs communs de base
dir, lang, style, title, width	attributs communs lies à HTML
onclick, ondblclick, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup	attributs communs lies aux événements Javascript

Le tag <dataTable> parcours les données et pour chaque occurrence, il créé une ligne dans le tableau.

L'attribut value représente une expression qui précise les données à utiliser. Ces données peuvent être sous la forme :

- d'un tableau
- d'un objet de type java.util.List
- d'un objet de type java.sql.ResultSet
- d'un objet de type javax.servlet.jsp.jstl.sql.Result
- d'un objet de type javax.faces.model.DataModel

Pour chaque élément encapsulés dans les données, le tag dataTable créé une nouvelle ligne.

Quelque soit le type qui encapsule les données, le composant dataTable va les mapper dans un objet de type DataModel. C'est cet objet que le composant va utiliser comme source de données. JSF définit 5 classes qui héritent de la classe DataModel : ArrayDataModel, ListDataModel, ResultDataModel, ResultSetDataModel et ScalarDataModel.

La méthode getWrappedObject() permet d'obtenir la source de données fournie en paramètre de l'attribut value.

L'attribut item permet de préciser le nom d'une variable qui va contenir les données d'une occurrence.

Chaque colonne est définie grâce à un tag <column>.

Exemple :

```
<h:dataTable value="#{listePersonnes.personneItems}" var="personne" cellspacing="4">
  <h:column>
    <f:facet name="header">
      <h:outputText value="Nom" />
    </f:facet>
    <h:outputText value="#{personne.nom}"/>
  </h:column>

  <h:column>
    <f:facet name="header">
      <h:outputText value="Prenom" />
    </f:facet>
    <h:outputText value="#{personne.prenom}" />
  </h:column>

  <h:column>
    <f:facet name="header">
      <h:outputText value="Date de naissance" />
    </f:facet>
    <h:outputText value="#{personne.datenaiss}"/>
  </h:column>

  <h:column>
    <f:facet name="header">
      <h:outputText value="Poids" />
    </f:facet>
    <h:outputText value="#{personne.poids}"/>
```

```
  </h:column>

  <h:column>
    <f:facet name="header">
      <h:outputText value="Taille" />
    </f:facet>
    <h:outputText value="#{personne.taille}"/>
  </h:column>

</h:dataTable>
```

L'en-tête et le pied du tableau sont précisés avec un tag <facet> pour chacun dans chaque tag <column>

Dans l'exemple précédent l'instance listePersonnes est une classe dont le code est le suivant :

Exemple :

```
package com.jmd.test.jsf;

import java.util.ArrayList;
import java.util.Calendar;
import java.util.GregorianCalendar;
import java.util.List;

public class PersonnesBean {

  private List PersonneItems = null;

  public List getPersonneItems() {
    if (PersonneItems == null) {
      PersonneItems = new ArrayList();
      PersonneItems.add(new Personne("Nom1", "Prenom1",
            new GregorianCalendar(1967, Calendar.OCTOBER, 22).getTime(),10,1.10f));
      PersonneItems.add(new Personne("Nom2", "Prenom2",
            new GregorianCalendar(1972, Calendar.MARCH, 10).getTime(),20,1.20f));
      PersonneItems.add(new Personne("Nom3", "Prenom3",
            new GregorianCalendar(1944, Calendar.NOVEMBER, 4).getTime(),30,1.30f));
      PersonneItems.add(new Personne("Nom4", "Prenom4",
            new GregorianCalendar(1958, Calendar.JULY, 19).getTime(),40,1.40f));
      PersonneItems.add(new Personne("Nom5", "Prenom5",
            new GregorianCalendar(1934, Calendar.JANUARY, 6).getTime(),50,1.50f));
      PersonneItems.add(new Personne("Nom6", "Prenom6",
            new GregorianCalendar(1989, Calendar.DECEMBER, 12).getTime(),60,1.60f));
    }
    return PersonneItems;
  }

}
```

La méthode getPersonneItems() renvoie une collection d'objets de type Personne.

La classe Personne encapsule simplement les données d'une personne.

Exemple :

```
package com.jmd.test.jsf;

import java.util.Date;

public class Personne {
  private String nom;
  private String prenom;
  private Date datenaiss;
  private int poids;
  private float taille;
  private boolean supprime;

  public Personne(String nom, String prenom, Date datenaiss, int poids,
```

```
    float taille) {
    super();
    this.nom = nom;
    this.prenom = prenom;
    this.datenaiss = datenaiss;
    this.poids = poids;
    this.taille = taille;
    this.supprime = false;
  }

  public boolean isSupprime() {
    return supprime;
  }

  public void setSupprime(boolean supprimer) {
    supprime = supprimer;
  }

  public Date getDatenaiss() {
    return datenaiss;
  }

  public void setDatenaiss(Date datenaiss) {
    this.datenaiss = datenaiss;
  }

  public String getNom() {
    return nom;
  }

  public void setNom(String nom) {
    this.nom = nom;
  }

  public int getPoids() {
    return poids;
  }

  public void setPoids(int poids) {
    this.poids = poids;
  }

  public String getPrenom() {
    return prenom;
  }

  public void setPrenom(String prenom) {
    this.prenom = prenom;
  }

  public float getTaille() {
    return taille;
  }

  public void setTaille(float taille) {
    this.taille = taille;
  }
}
```

Il est très facile de préciser un style particulier pour des lignes paires et impaires.

Il suffit de définir les deux styles désirés.

Exemple dans la partie en-tête de la JSP :

```
<STYLE type="text/css">
<!--
.titre {
background-color:#000000;
color:#FFFFFF;
}
```

```
.paire {
background-color:#EFEFEF;
}
.impaire {
background-color:#CECECE;
}
-->
</STYLE>
```

Il suffit d'utiliser les attributs headerClass, footerClass, rowClasses ou columnClasses. Avec ces deux derniers attributs, il est possible de préciser plusieurs style séparés par une virgule pour définir le style de chacune des lignes de façons répétitives.

Exemple :

```
<h:dataTable value="#{listePersonnes.personneItems}" var="personne"
  cellspacing="4" width="60%" rowClasses="paire,impaire" headerClass="titre">
```

Résultat :

Nom	Prenom	Date de naissance	Poids	Taille
Nom1	Prenom1	22/10/1967	10	1.1
Nom2	Prenom2	10/03/1972	20	1.2
Nom3	Prenom3	04/11/1944	30	1.3
Nom4	Prenom4	19/07/1958	40	1.4
Nom5	Prenom5	06/01/1934	50	1.5
Nom6	Prenom6	12/12/1989	60	1.6

Les éléments du tableau peuvent par exemple être sélectionnés grâce à une case à cocher pour permettre de réaliser des traitements sur les éléments marqués.

Il suffit de rajouter dans l'exemple précédent une colonne contenant une case et cocher et ajouter un bouton qui va réaliser les traitements sur les éléments cochés.

Exemple dans la JSP :

```
...
<h:form>
  <h1>Test</H1>
  <div align="center">
    <h:dataTable value="#{listePersonnes.personneItems}" var="personne"
          cellspacing="4" width="60%" rowClasses="paire,impaire" headerClass="titre">
...
    <h:column>
      <f:facet name="header">
        <h:outputText value="Sélection"/>
      </f:facet>
      <h:selectBooleanCheckbox value="#{personne.supprime}" />
    </h:column>

    </h:dataTable>

    <p>
      <h:commandButton value="Supprimer les sélectionnés"
        action="#{listePersonnes.supprimer}"/>
    </p>

  </div>
</h:form>
...
```

Il suffit alors d'ajouter les traitements dans la méthode supprimer() de la classe PersonnesBean qui sera appelée lors d'un clic sur le bouton « Supprimer les sélectionnés ».

Exemple :

```
public class PersonnesBean {

...

  public String supprimer() {
    Iterator iterator = personneItems.iterator();
    Personne pers=null;
    while (iterator.hasNext()) {
      pers = (Personne) iterator.next();
      System.out.println("nom="+pers.getNom()+" "+pers.isSupprime());
      // ajouter les traitements utiles
    }
    return null;
  }
}
```

Nom	Prenom	Date de naissance	Poids	Taille	Sélection
Nom1	Prenom1	22/10/1967	10	1.1	☐
Nom2	Prenom2	10/03/1972	20	1.2	☑
Nom3	Prenom3	04/11/1944	30	1.3	☐
Nom4	Prenom4	19/07/1958	40	1.4	☑
Nom5	Prenom5	06/01/1934	50	1.5	☐
Nom6	Prenom6	12/12/1989	60	1.6	☑

Supprimer les sélectionnés

Un clic sur le bouton « Supprimer les sélectionnés » affiche dans la console, la liste des éléments avec l'état de la case à cocher.

Exemple :

```
nom=Nom1 false
nom=Nom2 true
nom=Nom3 false
nom=Nom4 true
nom=Nom5 false
nom=Nom6 true
```

59.10. La gestion et le stockage des données

Les données sont stockées dans un ou plusieurs java bean qui encapsulent les différentes données des composants.

Ces données possèdent deux représentations :

- une contenue en interne par le modèle
- une pour leur présentation dans l'interface graphique (pour la saisie ou l'affichage)

Chaque objet de type Renderer possède une représentation par défaut des données. La transformation d'une représentation en une autre est assurée par des objets de type Converter. JSF fourni en standard plusieurs objets de type

Converter mais il est aussi possible de développer ces propres objets.

59.11. La conversion des données

JSF propose en standard un mécanisme de conversion des données. Celui ci repose sur un ensemble de classes dont certaines sont fournis en standard pour des conversions de base. Il est possible de définir ces propres classes de conversion pour répondre à des besoins spécifiques.

Ces conversions sont nécessaires car toutes les données transmises et affichées le sont sous la forme de chaîne de caractères. Cependant, leur exploitation dans les traitements nécessite souvent qu'elles soient stockées dans un autre format pour être exploité : un exemple flagrant est une données de type date.

Toutes les données saisies par l'utilisateur sont envoyées dans la requête http sont sous la forme de chaînes de caractères. Chacune de ces valeurs est désignée par « request value » dans les spécifications de JSF.

Ces valeurs sont stockées dans leur composant respectif dans des champs désignés par « submitted value » dans les spécifications.

Ces valeurs sont ensuite éventuellement converties implicitement ou explicitement et sont stockées dans leur composant respectif dans des champs désignés par « local value ». Ces données sont ensuite éventuellement validées.

L'intérêt d'un tel procédé est de s'assurer que les données seront valides avant de pouvoir les utiliser dans les traitements. Si la conversion ou la validation échoue, les traitements du cyle de vie de la page sont arrêtés et la page est réaffichée pour permettre l'affichage de messages d'erreurs. Sinon la phase de mise à jour des données (« Update model values ») du modèle est exécutée.

Les spécifications JSF imposent l'implémentation des convertisseurs suivants : javax.faces.DateTime, javax.faces.Number, javax.faces.Boolean, javax.faces.Byte, javax.faces.Character, javax.faces.Double, javax.faces.Float, javax.faces.Integer, javax.faces.Long, javax.faces.Short, javax.faces.BigDecimal et javax.faces.BigInteger.

JSF effectue une conversion implicite des données lorsque celle ci correspond à un type primitif ou à BigDecimal ou BigInteger en utilisant les convertisseurs appropriés.

Deux convertisseurs sont proposés en standard pour mettre en oeuvre des conversions qui ne correspondent pas à des types primitifs :

- le tag convertNumber : utilise le convertisseur javax.faces.Number
- le tag convertDateTime : utilise le convertisseur javax.faces.DateTime

59.11.1. Le tag <convertNumber>

Ce tag permet d'ajouter à un composant un convertisseur de valeur numérique.

Ce tag possède les attributs suivants :

Attributs	Rôle
type	type de valeur. Les valeurs possibles sont number (par défaut), currency et percent
pattern	motif de formatage qui sera utilisé par une instance de java.text.DecimalFormat
maxFractionDigits	nombre maximum de chiffres composant la partie décimale
minFractionDigits	nombre minimum de chiffres composant la partie décimale
maxIntegerDigits	nombre maximum de chiffres composant la partie entière
minIntegerDigits	nombre minimum de chiffres composant la partie entière

integerOnly	booléen qui précise si uniquement la partie entière est prise en compte (false par défaut)
groupingUsed	booléen qui précise si le séparateur de groupe d'unité est utilisé (true par défaut)
locale	objet de type java.util.Locale permettant de définir la locale à utiliser pour les conversions
currencyCode	code de la monnaie utilisée pour la conversion
currencySymbol	symbole de la monnaie utilisé pour la conversion

Exemple :

```
<p>valeur1 = <h:outputText value="#{convert.prix}">
<f:convertNumber type="currency"/>
</h:outputText>
</p>

<p>valeur2 = <h:outputText value="#{convert.poids}">
<f:convertNumber type="number"/>
</h:outputText>
</p>

<p>valeur3 = <h:outputText value="#{convert.ratio}">
<f:convertNumber type="percent"/>
</h:outputText>
</p>

<p>valeur4 = <h:outputText value="#{convert.prix}">
<f:convertNumber integerOnly="true" maxIntegerDigits="2"/>
</h:outputText>
</p>

<p>valeur5 = <h:outputText value="#{convert.prix}">
<f:convertNumber pattern="#.##"/>
</h:outputText>
</p>
```

Le code bean utilisé dans cet exemple est le suivant :

Exemple :

```
package com.jmd.test.jsf;

public class Convert {
  private int poids;
  private float prix;
  private float ratio;

  public Convert() {
    super();
    this.poids = 12345;
    this.prix = 1234.56f ;
    this.ratio = 0.12f ;
  }


  public int getPoids() {
    return poids;
  }

  public void setPoids(int poids) {
    this.poids = poids;
  }
  public float getRatio() {
    return ratio;
  }

  public void setRatio(float ratio) {
    this.ratio = ratio;
  }
```

```
  public float getPrix() {
    return prix;
  }

  public void setPrix(float prix) {
    this.prix = prix;
  }
}
```

valeur1 = 1 234,56 €

valeur2 = 12 345

valeur3 = 12%

valeur4 = 34,56

valeur5 = 1234,56

59.11.2. Le tag <convertDateTime>

Ce tag permet d'ajouter à un composant un convertisseur de valeurs temporelles.

Ce tag possède les attributs suivants :

Attributs	Rôle
Type	type de valeur. Les valeurs possibles sont date (par défaut), time et both
dateStyle	style prédéfini de la date. Les valeurs possibles sont short, medium, long, full ou default
timeStyle	style prédéfini de l'heure. Les valeurs possibles sont short, medium, long, full ou default
Pattern	motif de formatage qui sera utilisé par une instance de java.text.SimpleDateFormat
Locale	objet de type java.util.Locale permettant de définir la locale à utiliser pour les conversions
timeZone	objet de type java.util.TimeZone utilisé lors des conversions

Exemple :

```
<p>Date1 = <h:outputText value="#{convertDate.dateNaiss}">
<f:convertDateTime pattern="MM/yyyy"/>
</h:outputText>
</p>

<p>Date2 = <h:outputText value="#{convertDate.dateNaiss}">
<f:convertDateTime pattern="EEE, dd MMM yyyy"/>
</h:outputText>
</p>

<p>Date3 = <h:outputText value="#{convertDate.dateNaiss}">
<f:convertDateTime pattern="dd/MM/yyyy"/>
</h:outputText>
</p>

<p>Date4 = <h:outputText value="#{convertDate.dateNaiss}">
<f:convertDateTime dateStyle="full"/>
</h:outputText>
</p>
```

Le code du bean utilisé comme source dans cet exemple est le suivant :

Exemple :

```
package com.jmd.test.jsf;

import java.util.Date;

public class ConvertDate {
  private Date dateNaiss;

  public ConvertDate() {
    super();
    this.dateNaiss = new Date();
  }

  public Date getDateNaiss() {
    return dateNaiss;
  }

  public void setDateNaiss(Date dateNaiss) {
    this.dateNaiss = dateNaiss;
  }
}
```

Résultat :

Date1 = 06/2005

Date2 = mer., 15 juin 2005

Date3 = 15/06/2005

Date4 = mercredi 15 juin 2005

59.11.3. L'affichage des erreurs de conversions

Les messages d'erreurs issus de ces conversions peuvent être affichés en utilisant les tag <message> ou <messages>.

Par défaut, ils contiennent une description : « Conversion error occured ».

Pour modifier ce message par défaut ou l'internationaliser, il faut définir une clé javax.faces.component.UIInput.CONVERSION dans le fichier properties de définition des chaînes de caractères.

Exemple :

```
javax.faces.component.UIInput.CONVERSION=La valeur saisie n'est pas correctement formatée.
```

59.11.4. L'écriture de convertisseurs personnalisés

JSF fournit en standard des convertisseurs pour les types primitifs et quelques objets de base. Il peut être nécessaire de développer son propre convertisseur pour des besoins spécifiques.

Pour écrire son propre convertisseur, il faut définir une classe qui implémente l'interface Converter. Cette interface définit deux méthodes :

* Object getAsObject(FacesContext context, UIComponent component, String newValue) : cette méthode permet de convertir une chaîne de caractères en objet
* String getAsString(FacesContext context, UIComponent component, Object value) : cette méthode permet de convertir un objet en chaîne de caractères

La méthode getAsObject() doit lever une exception de type ConverterException si une erreur de conversion est détectée dans les traitements.

La suite de ce chapitre sera développée dans une version future de ce document

59.12. La validation des données

JSF propose en standard un mécanisme de validation des données. Celui ci repose sur un ensemble de classes qui permettent de faire des vérifications standards. Il est possible de définir ces propres classes de validation pour répondre à des besoins spécifiques.

La validation peut se faire de deux façons : au niveau de certains composants ou avec des classes spécialement développées pour des besoins spécifiques. Ces classes sont attachables à un composant et sont réutilisables. Ces validations sont effectuées côté serveur.

Les validators sont enregistrés sur des composants. Ce sont des classes qui utilisent des données pour effectuer des opérations de validation de la valeur des données : contrôle de présence, de type de données, de plage de valeurs, de format, ...

59.12.1. Les classes de validation standard

Toutes ces classes implémentent l'interface javax.faces.validator.Validator. JSF propose en standard plusieurs classes pour la validation :

* deux classes de validation sur une plage de données : LongRangeValidator et DoubleRangeValidator
* une classe de validation de la taille d'une chaîne de caractères : LengthValidator

Pour faciliter l'utilisation de ces classes, la bibliothèque de tags personnalisées core propose des tags dédiés à la mise en oeuvre de ces classes :

* validateDoubleRange : utilise la classe DoubleRangeValidator
* validateLongRange : utilise la classe LongRangeValidator
* validateLength : utilise la classe LengthValidator

Ces trois tags possèdent deux attributs nommés minimum et maximum qui permettent de préciser respectivement la valeur de début et de fin selon le Validator utilisé. L'un, l'autre ou les deux attributs peuvent être utilisés.

L'ajout d'une validation sur un contrôle peut se faire de plusieurs manières :

* ajout d'une ou plusieurs validations directement dans la JSP
* ajout par programmation d'une validation en utilisant la méthode addValidator().
* certaines implémentations de composants peuvent contenir des validations implicites.

Pour ajouter une validation à un composant dans la JSP , il suffit d'insérer le tag de validation dans le corps du tag du composant.

Exemple :

```
<h:inputText id="nombre" converter="#{Integer}" required="true"
  value="#{saisieDonnees.nombre}">
  <f:validate_longrange minimum="1" maximum="9" />
</h:inputText>
```

Certaines implémentations de composants peuvent contenir des validations implicites en fonction du contexte. C'est par exemple le cas du composant <inputText> qui, lorsque que son attribut required est à true, effectue un contrôle de présence de données saisies.

Exemple :

```
<h:inputText id="nombre" converter="#{Integer}" required="true"
  value="#{saisieDonnees.nombre}"/>
```

Toutes les validations sont faites côté serveur dans la version courante de JSF.

Les messages d'erreurs issus de ces conversions peuvent être affichés en utilisant les tags <message> ou <messages>.

Ils contiennent une description par défaut selon le validator utilisé commençant par « Validation error : ».

Pour modifier ce message par défaut ou l'internationaliser, il faut définir une clé dédiée dans le fichier properties de définition des chaînes de caractères. Les clés définies sont les suivantes :

- javax.faces.component.UIInput.REQUIRED
- javax.faces.validator.NOT_IN_RANGE
- javax.faces.validator.DoubleRangeValidator.MAXIMUM
- javax.faces.validator.DoubleRangeValidator.TYPE
- javax.faces.validator.DoubleRangeValidator.MINIMUM
- javax.faces.validator.LongRangeValidator.MAXIMUM
- javax.faces.validator.LongRangeValidator.MINIMUM
- javax.faces.validator.LongRangeValidator.TYPE
- javax.faces.validator.LengthValidator.MAXIMUM
- javax.faces.validator.LengthValidator.MINIMUM

59.12.2. Contourner la validation

Dans certains cas, il est nécessaire d'empêcher la validation. Par exemple, dans une page de saisie d'informations disposant d'un bouton « Valider » et « Annuler ». La validation doit être opérée lors d'un clic sur le bouton « Valider » mais ne doit pas l'être lors d'un clic sur le bouton « Annuler ».

Pour chaque composant dont l'action doit être exécutée sans validation, il faut mettre l'attribut immediate du composant à true.

Exemple :

```
<h:commandButton value="Annuler" action="annuler" immediate="true"/>
```

59.12.3. L'écriture de classes de validation personnalisées

JSF fournit en standard des classes de validation de base. Il peut être nécessaire de développer ses propres classes de validation pour des besoins spécifiques.

Pour écrire sa propre classe de validation, il faut définir une classe qui implémente l'interface javax.faces.validator.Validator. Cette interface définit une seule méthode :

* public void validate(FacesContext context, UIComponent component, Object toValidate) : cette méthode permet de réaliser les traitements de validation

Elle attend en paramètre :

* un objet de type FacesContext qui permet d'accéder au contexte de l'application jsf
* un objet de type UICOmponent qui contient une référence sur le composant dont la donnée est à valider
* un objet de type Object qui encapsule la valeur de la données à valider.

La méthode validate() doit levée une exception de type ValidatorException si une erreur dans les traitements de validation est détectée.

Exemple :

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.component.UIInput;
import javax.faces.context.FacesContext;
import javax.faces.validator.Validator;
import javax.faces.validator.ValidatorException;

import com.sun.faces.util.MessageFactory;

public class NumeroDeSerieValidator implements Validator {

  public static final String CLE_MESSAGE_VALIDATION_IMPOSSIBLE =
    "message.validation.impossible";

  public void validate(FacesContext contexte, UIComponent composant,
    Object objet) throws ValidatorException {
    String valeur = null;
    boolean estValide = false;

    if ((contexte == null) || (composant == null)) {
      throw new NullPointerException();
    }
    if (!(composant instanceof UIInput)) {
      return;
    }

    valeur = objet.toString();

    Pattern p = Pattern.compile("[0-9][0-9]-[0-9][0-9][0-9]",Pattern.MULTILINE);
    Matcher m = p.matcher(valeur);
    estValide = m.matches();

    if (!estValide) {
      FacesMessage errMsg = MessageFactory.getMessage(contexte,
      CLE_MESSAGE_VALIDATION_IMPOSSIBLE);
      throw new ValidatorException(errMsg);
    }

  }
}
```

Dans l'exemple précédent, la valeur à valider doit respecter une expression régulière de la forme deux chiffres, un tiret et trois chiffres.

Si la validation échoue alors il sera nécessaire d'informer l'utilisateur de la raison de l'échec grâce à un message stocké dans le resourceBundle de l'application.

Exemple :

```
message.validation.impossible=Le format du numéro de série est erroné
```

La valeur du message dans le resourceBundle peut être obtenue en utilisant la méthode getMessage() de la classe MessageFactory. Cette méthode attend en paramètres le contexte JSF de l'application et la clé du ressourceBundle à extraire. Elle renvoie un objet de type FacesMessages. Il suffit alors simplement de fournir cet objet à la nouvelle instance de la classe ValidatorException.

Pour pouvoir utiliser une classe de validation, il faut la déclarer dans le fichier de configuration.

Exemple :

```
<validator>
  <validator-id>com.jmd.test.jsf.NumeroDeSerie</validator-id>
  <validator-class>com.jmd.test.jsf.NumeroDeSerieValidator</validator-class>
</validator>
```

Le tag <validator-id> permet de définir un identifiant pour la classe de validation. Le tag <validator-class> permet de préciser la classe pleinement qualifiée.

Pour utiliser la classe de validation dans une page, il faut utiliser le tag <validator> en fournissant à l'attribut validatorId la valeur donnée au tag <validator-id> dans le fichier de configuration :

Exemple :

```
<h:panelGrid columns="2">
  <h:outputText value="Numéro de série : " />
  <h:panelGroup>
    <h:inputText value="#{validation.numeroSerie}" id="numeroSerie" required="true">
      <f:validator validatorId="com.jmd.test.jsf.NumeroDeSerie"/>
    </h:inputText>
    <h:message for="numeroSerie"/>
  </h:panelGroup>
</h:panelGrid>
```

La saisie d'un numéro répondant à l'expression régulière et l'appui sur la touche entrée n'affiche aucun message d'erreur :

La saisie d'un numéro ne répondant pas à l'expression régulière affiche le message d'erreur :

59.12.4. La validation à l'aide de bean

Il est possible de définir une méthode dans un bean qui va offrir les services de validation. Cette méthode doit avoir une signature similaire à celle de la méthode validate() de l'interface Validator.

Exemple :

```
package com.jmd.test.jsf;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.component.UIInput;
import javax.faces.context.FacesContext;
import javax.faces.validator.ValidatorException;

import com.sun.faces.util.MessageFactory;

public class Validation {
  public static final String CLE_MESSAGE_VALIDATION_IMPOSSIBLE =
    "message.validation.impossible";

  private String numeroSerie;

  public String getNumeroSerie() {
    return numeroSerie;
  }

  public void setNumeroSerie(String numeroSerie) {
    this.numeroSerie = numeroSerie;
  }

  public void valider(FacesContext contexte, UIComponent composant, Object objet) {
    String valeur = null;
    boolean estValide = false;

    if ((contexte == null) || (composant == null)) {
      throw new NullPointerException();
    }
    if (!(composant instanceof UIInput)) {
      return;
    }

    valeur = objet.toString();

    Pattern p = Pattern.compile("[0-9][0-9]-[0-9][0-9][0-9]",Pattern.MULTILINE);
    Matcher m = p.matcher(valeur);
    estValide = m.matches();

    if (!estValide) {
      FacesMessage errMsg = MessageFactory.getMessage(contexte,
      CLE_MESSAGE_VALIDATION_IMPOSSIBLE);
      throw new ValidatorException(errMsg);
    }
  }
}
```

Pour utiliser cette méthode, il faut utiliser l'attribut validator et lui fournir en paramètre une expression qui désigne la méthode d'une instance du bean

Exemple :

```
<h:panelGrid columns="2">
  <h:outputText value="Numéro de série : " />
  <h:panelGroup>
    <h:inputText value="#{validation.numeroSerie}" id="numeroSerie"
      required="true" validator="#{validation.valider}" />
    <h:message for="numeroSerie"/>
```

```
  </h:panelGroup>
</h:panelGrid>
```

Cette approche est particulièrement utile pour des besoins spécifiques à une application car sa mise en oeuvre est difficilement portable d'une application à une autre.

59.12.5. La validation entre plusieurs composants

De base, le modèle de validation des données proposé par JSF repose sur une validation unitaire de chaque composant. Il est cependant fréquent d'avoir besoin de faire une validation en fonction des données d'un ou plusieurs autres composants.

Pour réaliser ce genre de tâche, il faut définir un backing bean qui aura accès à chacun des composants nécessaire aux traitements et de définir dans ce bean une méthode qui va réaliser les traitements de validation.

Exemple :

```
package com.jmd.test.jsf;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.component.UIInput;
import javax.faces.context.FacesContext;
import javax.faces.validator.ValidatorException;

import com.sun.faces.util.MessageFactory;

public class Validation {
  public static final String CLE_MESSAGE_VALIDATION_IMPOSSIBLE =
    "message.validation.impossible";

  private String numeroSerie;
  private String cle;
  private UIInput cleInput;
  private UIInput numeroSerieInput;

  public String getNumeroSerie() {
    return numeroSerie;
  }

  public void setNumeroSerie(String numeroSerie) {
    this.numeroSerie = numeroSerie;
  }
  public void valider(FacesContext contexte, UIComponent composant, Object objet) {
    String valeur = null;
    boolean estValide = false;

    if ((contexte == null) || (composant == null)) {
      throw new NullPointerException();
    }
    if (!(composant instanceof UIInput)) {
      return;
    }

    valeur = objet.toString();

    Pattern p = Pattern.compile("[0-9][0-9]-[0-9][0-9][0-9]",Pattern.MULTILINE);
    Matcher m = p.matcher(valeur);
    estValide = m.matches();

    if (!estValide) {
      FacesMessage errMsg = MessageFactory.getMessage(contexte,
      CLE_MESSAGE_VALIDATION_IMPOSSIBLE);
      throw new ValidatorException(errMsg);
```

```
    }
  }

  public void validerCle(FacesContext contexte, UIComponent composant, Object objet) {
    System.out.println("validerCle");

    String valeurNumero = numeroSerieInput.getLocalValue().toString();
    String valeurCle = cleInput.getLocalValue().toString();
    boolean estValide = false;
    if (contexte == null) {
      throw new NullPointerException();
    }

    Pattern p = Pattern.compile("[0-9][0-9]-[0-9][0-9][0-9]",Pattern.MULTILINE);
    Matcher m = p.matcher(valeurNumero);
    estValide = m.matches() && valeurCle.equals("789");

    System.out.println("estValide="+estValide);
    if (!estValide) {
      FacesMessage errMsg = MessageFactory.getMessage(contexte,
      CLE_MESSAGE_VALIDATION_IMPOSSIBLE);
      throw new ValidatorException(errMsg);
    }
  }

  public String getCle() {
    return cle;
  }

  public void setCle(String cle) {
    this.cle = cle;
  }

  public UIInput getCleInput() {
    return cleInput;
  }

  public void setCleInput(UIInput cleInput) {
    this.cleInput = cleInput;
  }

  public UIInput getNumeroSerieInput() {
    return numeroSerieInput;
  }

  public void setNumeroSerieInput(UIInput numeroSerieInput) {
    this.numeroSerieInput = numeroSerieInput;
  }
}
```

Il suffit alors d'ajouter un champ caché dans la vue sur lequel la classe de validation sera appliquée.

Exemple :

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ page language="java" %>
<!DOCTYPE HTML PUBLIC "-//w3c//dtd html 4.0 transitional//en">
<html>
<f:view>
<head>
  <title>Tests de validation</title>
</head>
<body bgcolor="#FFFFFF">
  <h:form>
  <h2>Tests de validation</h2>

  <h:panelGrid columns="2">
    <h:outputText value="Numéro de série : " />
    <h:panelGroup>
      <h:inputText value="#{validation.numeroSerie}" id="numeroSerie"
```

```
        required="true" binding="#{validation.numeroSerieInput}" />
      <h:message for="numeroSerie"/>
    </h:panelGroup>
    <h:outputText value="clé : " />
    <h:panelGroup>
      <h:inputText value="#{validation.cle}" id="cle" binding="#{validation.cleInput}"
        required="true" />
      <h:message for="validationCle"/>
    </h:panelGroup>
  </h:panelGrid>

  <h:inputHidden id="validationCle" validator="#{validation.validerCle}" value="nul"/>

  <h:commandButton value="Valider" action="submit"/>

  </h:form>
</body>
</f:view>
</html>
```

59.12.6. L'écriture de tags pour un convertisseur ou un valideur de données

L'écriture de tag personnalisé facilite l'utilisation d'un convertisseur ou d'un valideur et permet de leur fournir des paramètres.

Il faut définir une classe nommée handler qui va contenir les traitements du tag. Cette classe doit hériter d'une sous classe dédiée selon le type d'élément que va représenter le tag :

- ConverterTag : si le tag concerne un convertisseur
- ValidatorTag : si le tag concerne un validateur
- UIComponentTag et UIComponentBodyTag : si le tag concerne un composant

Le handler est un bean dont une propriété doit correspondre à chaque attribut défini dans le tag.

Pour pouvoir utiliser un tag personnalisé, il faut définir un fichier .tld

Ce fichier au format XML défini dans les spécifications des JSP permet de fournir des informations sur la bibliothèque de tags personnalisés notamment la version des spécifications utilisées et des informations sur chaque tag.

Enfin, il est nécessaire de déclarer l'utilisation de la bibliothèque de tags personnalisés dans la JSP.

59.12.6.1. L'écriture d'un tag personnalisé pour un convertisseur

Il faut définir un handler pour le tag qui est un bean qui hérite de la classe ConverterTag.

Il est important dans le constructeur du handler de faire un appel à la méthode setConverterId() en lui passant un id défini dans le fichier de configuration de l'application JSF.

Il faut redéfinir la méthode release() dont les traitements vont permettre de réinitialiser les propriétés de la classe. Ceci est important car l'implémentation utilisée pour utiliser un pool pour ces objets afin d'augmenter les performances. La méthode release() est dans ce cas utilisée pour recycler les instances du pool non utilisées.

Il faut ensuite redéfinir la méthode createConverter() qui va permettre la création d'une instance du converter en utilisant les éventuels valeurs des attributs du tag.

La valeur fournie à un attribut d'un tag pour être soit un littéral soit une expression dont le contenu devra être évalué pour connaître la valeur à un instant donné.

La suite de ce chapitre sera développée dans une version future de ce document

59.12.6.2. L'écriture d'un tag personnalisé pour un valideur

L'écriture d'un tag personnalisé pour un valideur suit les mêmes règles que pour un convertisseur. La grande différence est que la classe handler doit hériter de la classe ValidatorTag. La méthode à appeler dans le constructeur est la méthode setValidatorId() et la méthode à redéfinir pour créer une instance du valideur est la méthode createValidator().

La suite de ce chapitre sera développée dans une version future de ce document

59.13. La sauvegarde et la restauration de l'état

JSF sauvegarde l'état de chaque élément présent dans la vue : les composants, les convertisseurs, les valideurs, ... pourvu que ceux ci mettent en oeuvre un mécanisme adéquat.

Ces états sont stockés dans un champ de type hidden dans la vue pour permettre l'échange de ces informations entre deux requêtes si l'application est configurée dans ce sens dans le fichier de configuration.

Ce mécanisme peut prendre deux formes :

- la classe qui encapsule l'élément peut implémenter l'interface Serializable
- la classe qui encapsule l'élément peut implémenter l'interface StateHolder

Dans le premier cas, c'est le mécanisme standard de la sérialisation qui sera utilisé. Il nécessite donc très peux voir aucun code particulier si les champs de la classe sont tous d'un type qui est sérialisable.

L'implémentation de l'interface StateHolder nécessite la définition des deux méthodes définies dans l'interface

(saveState() et restoreState()) et la présence d'un constructeur par défaut. Cette approche peut être intéressante pour obtenir un contrôle très fin de la sauvegarde et de la restauration de l'état.

La méthode saveState(FacesContext) renvoie un objet sérialisable qui va contenir les données de l'état à sauvegarder. La méthode restoreState(FacesContext, Object) effectue l'opération inverse.

Il est aussi nécessaire de définir une propriété nommée transient de type booléen qui précise si l'état doit être sauvegardé ou non.

Si l'élément n'implémente pas l'interface Serializable ou StateHolder alors son état n'est pas sauvegardé entre deux échanges de la vue.

59.14. Le système de navigation

Une application de type web se compose d'un ensemble de pages dans lesquelles l'utilisateur navigue en fonction de ces actions.

Un système de navigation standard peut être facilement mis en oeuvre avec JSF grâce à un paramétrage au format XML dans le fichier de configuration de l'application.

Le système de navigation assure la gestion de l'enchainement des pages en utilisant des actions. Les règles de navigation sont des chaînes de caractères qui sont associés à une page d'origine et qui permet de déterminer la page de résultat. Toutes ces règles sont contenues dans le fichier de configuration face-config.xml.

La déclaration de ce système de navigation ressemble à celle utilisée dans le framework Struts.

Le système de navigation peut être statique ou dynamique. Dans ce dernier cas, des traitements particuliers doivent être mis en place pour déterminer la cible de la navigation.

Exemple :

```
...
  <navigation-rule>
    <from-view-id>/login.jsp</from-view-id>
    <navigation-case>
      <from-outcome>login</from-outcome>
      <to-view-id>/accueil.jsp</to-view-id>
    </navigation-case>
  </navigation-rule>
...
```

La tag <navigation-rule> permet de préciser des règles de navigation.

La tag <from-view-id> permet de préciser qu'elle est la page concernée. Ce tag n'est pas obligatoire : sans sa présence, il est possible définir une règle de navigation applicable à toutes les pages JSF de l'application.

Exemple :

```
<navigation-rule>
  <navigation-case>
    <from-outcome>logout</from-outcome>
    <to-view-id>/logout.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

Il est aussi possible de désigner un ensemble de page dans le tag <from-view-id> en utilisant le caractère * dans la valeur du tag. Ce caractère * ne peut être utilisé qu'une seule fois dans la valeur du tag et il doit être en dernière position.

Exemple :

```
<from-view-id>/admin/*</from-view-id>
```

Le tag <navigation-case> permet de définir les différents cas.

La valeur du tag <from-outcome> doit correspondre au nom d'une action.

Le tag <to-view-id> permet de préciser la page qui sera affichée. L'url fournie comme valeur doit commencer par un slash et doit préciser une page possédant une extension brute (ne surtout pas mettre une url utilisée par la servlet faisant office de contrôleur).

Le tag <redirect/> inséré juste après le tag <to-view-id> permet de demander la redirection vers la page au navigateur de l'utilisateur.

La gestion de la navigation est assurée par une instance de la classe NavigationHandler, gérée au niveau de l'application. Ce gestionnaire utilise la valeur d'un attribut action d'un composant pour déterminer la page suivante et faire la redirection vers la page adéquat en fonction des informations fournies dans le fichier de configuration.

La valeur de l'attribut action peut être statique : dans ce cas la valeur est en dur dans le code de la vue

Exemple :

```
<h:commandButton action="login"/>
```

La valeur de l'attribut action peut être dynamique : dans ce cas la valeur est déterminée par l'appel d'une méthode d'un bean

Exemple :

```
<h:commandButton action="#{login.verifierMotDePasse}"/>
```

Dans ce cas, la méthode appelée ne doit pas avoir de paramètres et doit retourner une chaîne de caractères définie dans la navigation du fichier de configuration.

Lors des traitements par le NavigationHandler, si aucune action ne trouve de correspondance dans le fichier de configuration pour la page alors la page est simplement réaffichée.

59.15. La gestion des événements

Le modèle de gestion de événements de JSF est similaire est celui utilisé dans les JavaBeans : il repose sur les Listener et les Event pour traiter les événements générés dans les composants graphiques suite aux actions de l'utilisateur.

Un objet de type Event encapsule le composant à l'origine de l'événement et des données relatives à cet événement.

Pour être notifié d'un événement particulier, il est nécessaire d'enregistrer un objet qui implémente l'interface Listener auprès du composant concerné.

Lors de certaines actions de l'utilisateur, un événement est émis.

L'implémentation JSF propose deux types d'événements :

- Value changed : ces événements sont émis lors du changement de la valeur d'un composant de type UIInput, UISelectOne, UISelectMany, et UISelectBoolean
- Action : ces événements sont émis lors d'un clic sur un hyperlien ou un bouton qui sont des composants de type UICommand

JSF propose de transposer le modèle de gestion des événements des interfaces graphiques des applications standalone aux applications de type web utilisant JSF.

La gestion des événements repose donc sur deux types d'objets

- Event : classe qui encapsule l'événement lui même
- Listener : classe qui va encapsuler les traitements à réaliser pour un type d'événements

Comme pour les interfaces graphiques des applications standalone, la classe de type Listener doit s'enregistrer auprès du composant concerné. Lorsque celui ci émet un événement suite à une action de l'utilisateur, il appelle le Listener enregistré en lui fournissant en paramètre un objet de type Event.

Exemple :

```
<h:selectOneMenu ... valueChangeListener="#{choixLangue.langueChangement}">
  ...
</h:selectOneMenu>
```

JSF supporte trois types d'événements :

- les changements de valeurs : concernent les composants qui permettent la saisie ou la sélection d'une valeur et que cette valeur change
- les actions : concernent un clic sur un bouton (commandButton) ou un lien (commandLink)
- les événements liés au cycle de vie : ils sont émis par le framework JSF durant le cyle de vie des traitements

Les traitements des listeners peuvent affecter la suite des traitements du cycle de vie de plusieurs manières :

- par défaut, laisser ces traitements se poursuivre
- demander l'exécution immédiate de la dernière étape en utilisant la méthode FacesContext.renderResponse()
- arrêter les traitements du cyle de vie en utilisant la méthode FacesContext.responseComplete()

59.15.1. Les événements liés à des changements de valeur

Il y a deux façons de préciser un listener de type valueChangeListener sur un composant :

- utiliser l'attribut valueChangeListener
- utiliser le tag valueChangeListener

L'attribut valueChangeListener permet de préciser une expression qui désigne une méthode qui sera exécutée durant les traitements du cyle de vie de la requête. Pour que ces traitements puissent être déclenchés, il faut soumettre la page.

Exemple :

```
<h:selectOneMenu value="#{choixLangue.langue}" onchange="submit()"
  valueChangeListener="#{choixLangue.langueChangement}">
  <f:selectItems value="#{choixLangue.langues}"/>
</h:selectOneMenu>
```

La méthode ne renvoie aucune valeur et attend en paramètre un objet de type ValueChangeEvent.

Exemple :

```
package com.jmd.test.jsf;

import java.util.Locale;
import javax.faces.context.FacesContext;
import javax.faces.event.ValueChangeEvent;
import javax.faces.model.SelectItem;

public class ChoixLangue {
```

```
  private static final String LANGUE_FR = "Français";
  private static final String LANGUE_EN = "Anglais";
  private String langue = LANGUE_FR;

  private SelectItem[] langueItems = {
    new SelectItem(LANGUE_FR, "Français"),
    new SelectItem(LANGUE_EN, "Anglais") };

  public SelectItem[] getLangues() {
    return langueItems;
  }

  public String getLangue() {
    return langue;
  }

  public void setLangue(String langue) {
    this.langue = langue;
  }

  public void langueChangement(ValueChangeEvent event) {
    FacesContext context = FacesContext.getCurrentInstance();
    System.out.println("Changement de la langue : "+event.getNewValue());
    if (LANGUE_FR.equals((String) event.getNewValue()))
      context.getViewRoot().setLocale(Locale.FRENCH);
    else
      context.getViewRoot().setLocale(Locale.ENGLISH);
    }
  }
}
```

La classe ValueChangeEvent possède plusieurs méthodes utiles :

Méthode	Rôle
UIComponent getComponent()	renvoie le composant qui a généré l'événement
Object getNewValue()	renvoie la nouvelle valeur (convertie et validée)
Object getOldValue()	renvoie la valeur précédente

Le tag valueChangeListener permet aussi de préciser un listener. Son attribut type permet de préciser une classe implémentant l'interface ValueChangeListener.

Exemple :

```
<h:selectOneMenu value="#{choixLangue.langue}" onchange="submit()">
  <f:valueChangeListener type="com.jmd.test.jsf.ChoixLangueListener"/>
  <f:selectItems value="#{choixLangue.langues}"/>
</h:selectOneMenu>
```

Une telle classe doit définir une méthode processValueChange() qui va contenir les traitements exécutés en réponse à l'événement.

Exemple :

```
package com.jmd.test.jsf;

import java.util.Locale;

import javax.faces.context.FacesContext;
import javax.faces.event.AbortProcessingException;
import javax.faces.event.ValueChangeEvent;
import javax.faces.event.ValueChangeListener;

public class ChoixLangueListener implements ValueChangeListener {
```

```
  private static final String LANGUE_FR = "Français";

  private static final String LANGUE_EN = "Anglais";

  public void processValueChange(ValueChangeEvent event)
    throws AbortProcessingException {
    FacesContext context = FacesContext.getCurrentInstance();
    System.out.println("Changement de la langue : " + event.getNewValue());
    if (LANGUE_FR.equals((String) event.getNewValue()))
      context.getViewRoot().setLocale(Locale.FRENCH);
    else
      context.getViewRoot().setLocale(Locale.ENGLISH);
    }
  }
}
```

59.15.2. Les événements liés à des actions

Les actions sont des clics sur des boutons ou des liens. Le clic sur un composant de type commandLink ou commandButton déclenche automatiquement la soumission de la page.

Il y a deux façons de préciser un listener de type actionListener sur un composant :

- utiliser l'attribut actionListener
- utiliser le tag actionListener

L'attribut actionListener permet de préciser une expression qui désigne une méthode qui sera exécutée durant les traitements du cyle de vie de la requête.

Exemple :

```
<table align="center" width="50%">
  <tr>
    <td width="50%"><h:commandButton image="images/bouton_valider.gif"
      actionListener="#{saisieDonnees.traiterAction}"
      id="Valider" />
    </td>
    <td><h:commandButton image="images/bouton_annuler.gif"
      actionListener="#{saisieDonnees.traiterAction}"
      id="Annuler"/>
    </td>
  </tr>
</table>
```

Cette méthode attend en paramètre un objet de type ActionEvent.

Exemple :

```
package com.jmd.test.jsf;

import javax.faces.context.FacesContext;
import javax.faces.event.ActionEvent;

public class SaisieDonnees {

  public void traiterAction(ActionEvent e) {

    FacesContext context = FacesContext.getCurrentInstance();

    String clientId = e.getComponent().getClientId(context);
    System.out.println("traiterAction : clientId=" + clientId);

  }
```

```
}
```

Le tag valueChangeListener permet aussi de préciser un listener. Son attribut type permet de préciser une classe implémentant l'interface ValueChangeListener.

Exemple :

```
<table align="center" width="50%">
  <tr>
    <td width="50%"><h:commandButton image="images/bouton_valider.gif" id="Valider" >
        <f:actionListener type="com.jmd.test.jsf.SaisieDonneesListener"/>
      </h:commandButton>
      </td>
    <td><h:commandButton image="images/bouton_annuler.gif" id="Annuler">
        <f:actionListener type="com.jmd.test.jsf.SaisieDonneesListener"/>
      </h:commandButton>
    </td>
  </tr>
</table>
```

Une telle classe doit définir la méthode processAction() définie dans l'interface.

Exemple :

```
package com.jmd.test.jsf;

import javax.faces.context.FacesContext;
import javax.faces.event.AbortProcessingException;
import javax.faces.event.ActionEvent;
import javax.faces.event.ActionListener;

public class SaisieDonneesListener implements ActionListener {

  public void processAction(ActionEvent e) throws AbortProcessingException {
    FacesContext context = FacesContext.getCurrentInstance();

    String clientId = e.getComponent().getClientId(context);
    System.out.println("processAction : clientId=" + clientId);

  }
}
```

59.15.3. L'attribut immediate

L'attribut immediate permet de demander les traitements immédiat des listeners.

Par exemple, sur une page un composant possède un attribut required et un second possède un listener. Les traitements du second doivent pouvoir être réalisés sans que le premier composant n'affiche un message d'erreur lié à sa validation.

Le cycle de traitement de la requête est modifié lorsque l'attribut immediate est positionné dans un composant. Dans ce cas, les données du composant sont converties et validées si nécessaire puis les traitements du listener sont exécutés à la place de l'étape « Process validations » (juste après l'étape Apply Request Value).

Exemple :

```
<h:selectOneMenu value="#{choixLangue.langue}" onchange="submit()" immediate="true">
  <f:valueChangeListener type="com.jmd.test.jsf.ChoixLangueListener"/>
  <f:selectItems value="#{choixLangue.langues}"/>
</h:selectOneMenu>
```

Par défaut, ceci modifie l'ordre d'exécution des traitements du cycle de vie mais n'empêche pour les traitements prévus de s'exécuter. Pour les inhiber, il est nécessaire de demande au framework JSF d'interrompre les traitements du cycle de vie en utilisant la méthode renderResponse() du context.

Exemple :

```
  public void langueChangement(ValueChangeEvent event) {
    FacesContext context = FacesContext.getCurrentInstance();
    System.out.println("Changement de la langue : " + event.getNewValue());
    if (LANGUE_FR.equals((String) event.getNewValue())) {
      context.getViewRoot().setLocale(Locale.FRENCH);
    } else {
      context.getViewRoot().setLocale(Locale.ENGLISH);
    }
    context.renderResponse();
  }
```

Le mode de fonctionnement est le même avec les actionListener hormis le fait que l'appel à la méthode renderResponse() est inutile puisqu'il est automatiquement fait par le framework.

59.15.4. Les événements liés au cycle de vie

Le framework émet des événements avant et après chaque étape du cycle de vie des requêtes. Ils sont traités par des phaseListeners.

L'enregistrement d'un phaseListener se fait dans le fichier de configuration dans un tag fils <phase-listener> fils du tag <lifecycle> qui doit contenir le nom pleinement qualifié d'une classe.

Exemple :

```
<faces-config>
...

  <lifecycle>
    <phase-listener>com.jmd.test.jsf.PhasesEcouteur</phase-listener>
  </lifecycle>

</faces-config>
```

La classe précisée doit implémenter l'interface javax.faces.event.PhaseListener qui définit trois méthodes :

- getPhaseId() : renvoie un objet de type PhaseId qui permet de préciser à quelle phase se listener correspond
- beforePhase() : traitements à exécuter avant l'éxécution de la phase
- afterPhase() : traitements à exécuter après l'exécution de la phase

La classe PhaseId définit des constantes permettant d'identifier chacune des phases : PhaseId.RESTORE_VIEW, PhaseId.APPLY_REQUEST_VALUES, PhaseId.PROCESS_VALIDATIONS, PhaseId.UPDATE_MODEL_VALUES, PhaseId.INVOKE_APPLICATION et PhaseId.RENDER_RESPONSE

Elle définit aussi la constante PhaseId.ANY_PHASE qui permet de demander l'application du listener à toutes les phases. Cela peut être très utile lors du débogage.

Exemple :

```
package com.jmd.test.jsf;

import javax.faces.event.PhaseEvent;
import javax.faces.event.PhaseId;
import javax.faces.event.PhaseListener;

public class PhasesEcouteur implements PhaseListener {
```

```
  public void afterPhase(PhaseEvent pe) {
    System.out.println("Apres " + pe.getPhaseId());
  }

  public void beforePhase(PhaseEvent pe) {
    System.out.println("Avant " + pe.getPhaseId());
  }

  public PhaseId getPhaseId() {
    return PhaseId.ANY_PHASE;
  }
}
```

Lors de l'appel de la première page de l'application, les informations suivantes sont affichées dans la sortie standard

Avant RESTORE_VIEW 1
Apres RESTORE_VIEW 1
Avant RENDER_RESPONSE 6
Apres RENDER_RESPONSE 6

Lors d'une soumission de cette page avec une erreur de validation des données, les informations suivantes sont affichées dans la sortie standard

Avant RESTORE_VIEW 1
Apres RESTORE_VIEW 1
Avant APPLY_REQUEST_VALUES 2
Apres APPLY_REQUEST_VALUES 2
Avant PROCESS_VALIDATIONS 3
Apres PROCESS_VALIDATIONS 3
Avant RENDER_RESPONSE 6
Apres RENDER_RESPONSE 6

Lors d'une soumission de cette page sans erreur de validation des données, les informations suivantes sont affichées dans la sortie standard

Avant RESTORE_VIEW 1
Apres RESTORE_VIEW 1
Avant APPLY_REQUEST_VALUES 2
Apres APPLY_REQUEST_VALUES 2
Avant PROCESS_VALIDATIONS 3
Apres PROCESS_VALIDATIONS 3
Avant UPDATE_MODEL_VALUES 4
Apres UPDATE_MODEL_VALUES 4
Avant INVOKE_APPLICATION 5
Apres INVOKE_APPLICATION 5
Avant RENDER_RESPONSE 6
Apres RENDER_RESPONSE 6

59.16. Le déploiement d'une application

Une application utilisant JSF s'exécute dans un serveur d'application contenant un conteneur web implémentant les spécifications servlet 1.3 et JSP 1.2 minimum. Une telle application doit être packagée dans un fichier .war.

La compilation des différentes classes de l'application nécessite l'ajout dans le classpath de la bibliothèque servlet.

Elle nécessite aussi l'ajout dans le classpath de la bibliothèque jsf-api.jar de la ou des bibliothèques requises par l'implémentation JSF utilisées.

Ces bibliothèques doivent aussi être disponibles pour le conteneur web qui va exécuter l'application. Le plus simple est de mettre ces fichiers dans le répertoire WEB-INF/lib

59.17. Un exemple d'application simple

Cette section va développer une petite application constituée de deux pages. La première va demander le nom de l'utilisateur et la seconde afficher un message de bienvenue.

Il faut créer un répertoire, par exemple nommé Test_JSF et créer à l'intérieur la structure de l'application qui correspond à la structure de toute application Web selon les spécifications J2EE, notamment le répertoire WEB-INF avec ces sous répertoires lib et classes.

Il faut ensuite copier les fichiers nécessaire à une utilisation de JSF dans l'application web.

Il suffit de copier*.jar du répertoire lib de l'implémentation de référence vers le répertoire WEB-INF/lib du projet.

Il faut créer un fichier à la racine du projet et le nommer index.htm

Exemple :

```
<html>
  <head>
    <meta http-equiv="Refresh" content= "0; URL=login.faces"/>
    <title>Demarrage de l'application</title>
  </head>
  <body>
    <p>D&eacute;marrage de l'application ...</p>
  </body>
</html>
```

Il faut créer un fichier à la racine du projet et le nommer login.jsp

Exemple :

```
<html>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<f:view>
<head>
  <title>Application de tests avec JSF</title>
</head>
<body>
  <h:form>
    <h3>Identification</h3>
    <table>
      <tr>
        <td>Nom : </td>
        <td><h:inputText value="#{login.nom}"/></td>
      </tr>
      <tr>
        <td>Mot de passe :</td>
        <td><h:inputSecret value="#{login.mdp}"/></td>
      </tr>
      <tr>
        <td colspan="2"><h:commandButton value="Login" action="login"/></td>
      </tr>
    </table>
  </h:form>
</body>
</f:view>
</html>
```

Il faut créer une nouvelle classe nommée com.jmd.test.jsf.LoginBean et la compiler dans le répertoire WEB-INF/classes.

Exemple :

```
package com.jmd.test.jsf;

public class LoginBean {

  private String nom;
  private String mdp;

  public String getMdp() {
    return mdp;
  }

  public String getNom() {
    return nom;
  }

  public void setMdp(String string) {
    mdp = string;
  }

  public void setNom(String string) {
    nom = string;
  }

}
```

Il faut créer un fichier à la racine du projet et le nommer accueil.jsp : cette page contiendra la page d'accueil de l'application.

Il faut créer un fichier dans le répertoire /WEB-INF et le nommer faces-config.xml

Exemple :

```
<?xml version="1.0"?>

<!DOCTYPE faces-config PUBLIC
"-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
"http://java.sun.com/dtd/web-facesconfig_1_0.dtd">

<faces-config>
  <navigation-rule>
    <from-view-id>/login.jsp</from-view-id>
    <navigation-case>
      <from-outcome>login</from-outcome>
      <to-view-id>/accueil.jsp</to-view-id>
    </navigation-case>
  </navigation-rule>

  <managed-bean>
    <managed-bean-name>login</managed-bean-name>
    <managed-bean-class>com.jmd.test.jsf.LoginBean</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
  </managed-bean>

</faces-config>
```

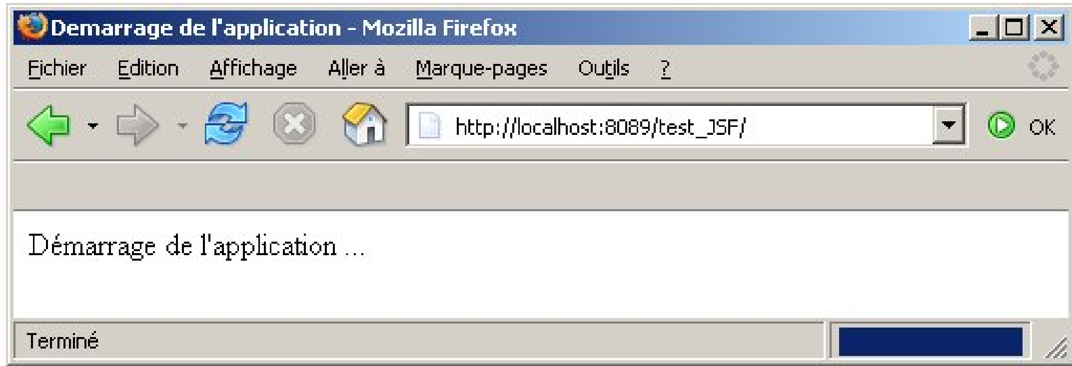
Il faut créer un fichier dans le répertoire /WEB-INF et le nommer web.xml

Exemple :

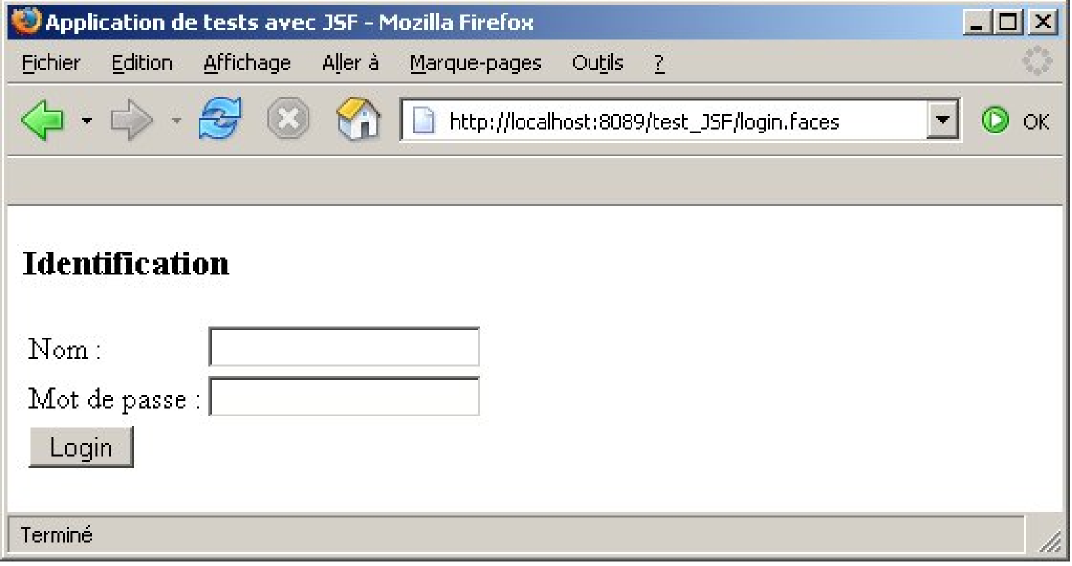
```
<?xml version="1.0"?>

<!DOCTYPE web-app PUBLIC
"-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
```

```
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.faces</url-pattern>
  </servlet-mapping>

  <welcome-file-list>
    <welcome-file>index.htm</welcome-file>
  </welcome-file-list>

</web-app>
```

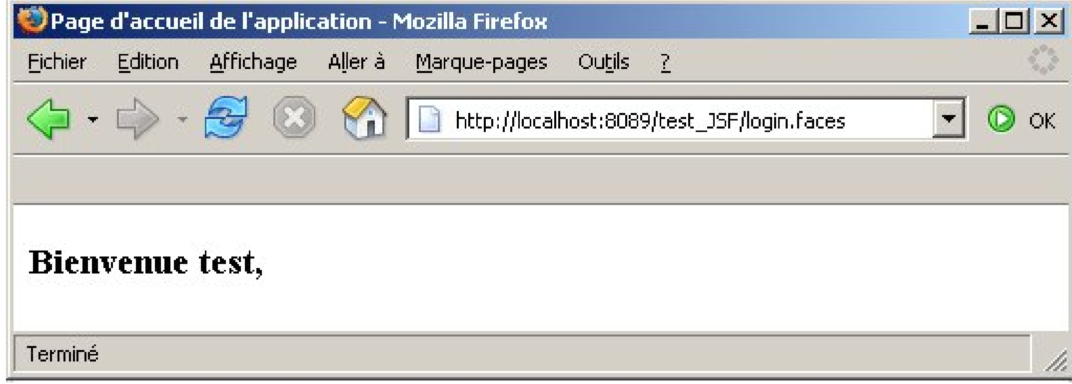
Il suffit alors de démarrer Tomcat, puis d'ouvrir un navigateur et taper l'url http://localhost:8089/test_JSF/ (en remplaçant le port 8089 par celui défini dans Tomcat).



Une fois l'application démarrée, la page de login s'affiche



Il faut saisir un nom par exemple test et cliquer sur le bouton « Login ».



Cette exemple ne met en aucune façon en valeur la puissance de JSF mais permet simplement de mettre en place les éléments minimum pour une application utilisant JSF.

59.18. L'internationalisation

JSF propose des fonctionnalités qui facilitent l'internationalisation d'une application.

Il faut définir un fichier au format properties qui va contenir la définition des chaînes de caractères. Un tel fichier possède les caractéristiques suivantes :

- le fichier doit avoir l'extension .properties
- il doit être dans le classpath de l'application
- il est composé d'une paire clé=valeur par ligne. La clé permet d'identifier de façon unique la chaîne de caractères

Exemple : le fichier msg.properties

```
login_titre=Application de tests avec JSF
login_identification=Identification
login_nom=Nom
login_mdp=Mot de passe
login_Login=Valider
```

Ce fichier correspond à la langue par défaut. Il est possible de définir d'autre fichier pour d'autres langues. Ces fichiers doivent être avoir le même nom suivi d'un underscore et du code langue défini par le standard ISO 639 avec toujours l'extension .properties.

Exemple :

```
msg.properties
msg_en.properties
msg_de.properties
```

Il faut bien sûre remplacer les valeurs de chaque chaîne par leur traduction correspondante.

Exemple :

```
login_titre=Tests of JSF
login_identification=Login
login_nom=Name
login_mdp=Password
login_Login=Login
```

Les langues disponibles doivent être précisées dans le fichier de configuration.

Exemple :

```
<faces-config>
...
  <application>
    <locale-config>
      <default-locale>fr</default-locale>
      <supported-locale>en</supported-locale>
    </locale-config>
  </application>
...
</faces-config>
```

Pour utiliser l'internationalisation dans les vues, il faut utiliser le tag <f:loadBundle> pour charger le fichier .properties nécessaire. Deux attributs de ce tags sont requis :

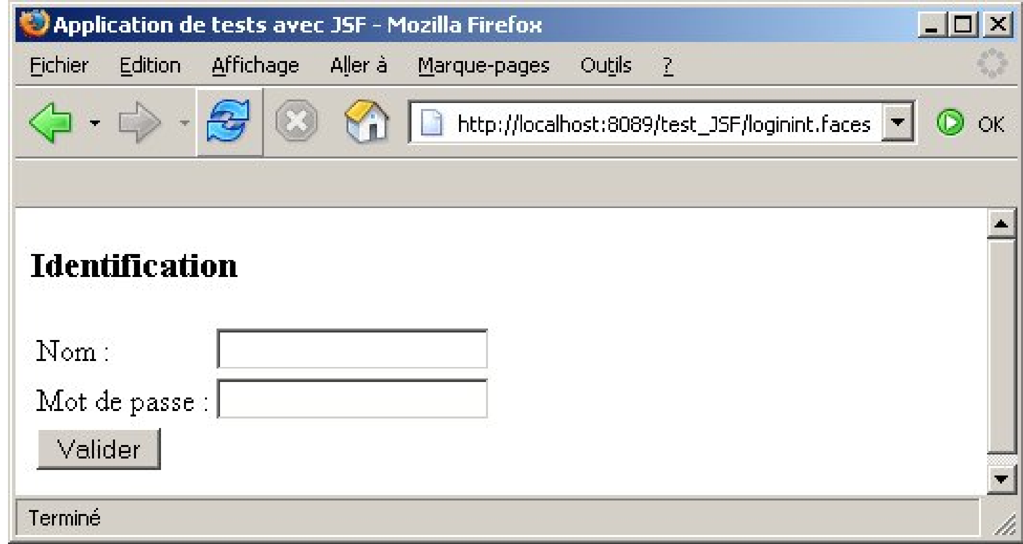
- basename : précise la localisation et le nom de base des fichiers .properties. La notation de la localisation est similaire à celle utilisée pour les packages
- var : précise le nom de la variable qui va contenir les chaînes de caractères

Il ne reste plus qu'à utiliser la variable définie en utilisant la notation avec un point pour la clé de la chaîne dont on souhaite utiliser la valeur.

Exemple :

```
<html>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<f:view>
<f:loadBundle basename="com.jmd.test.jsf.msg" var="msg"/>

<head>
  <title><h:outputText value="#{msg.login_titre}"/></title>
</head>
<body>
  <h:form>
    <h3><h:outputText value="#{msg.login_identification}"/></h3>
    <table>
      <tr>
        <td><h:outputText value="#{msg.login_nom}"/> : </td>
        <td><h:inputText value="#{login.nom}"/></td>
      </tr>
      <tr>
        <td><h:outputText value="#{msg.login_mdp}"/> :</td>
        <td><h:inputSecret value="#{login.mdp}"/></td>
      </tr>
      <tr>
        <td colspan="2"><h:commandButton value="#{msg.login_Login}" action="login"/></td>
      </tr>
    </table>
  </h:form>
</body>
</f:view>
</html>
```

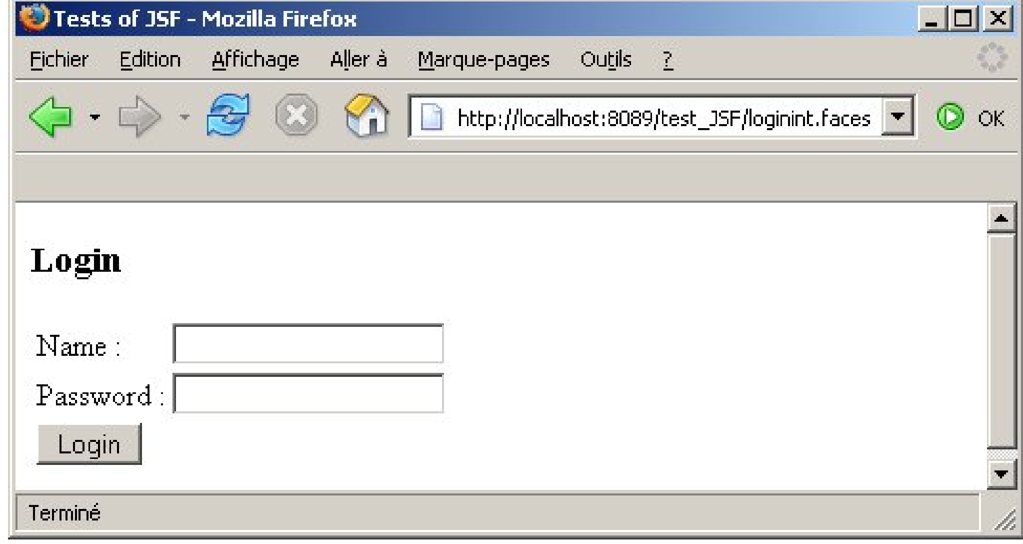
La langue à utiliser est déterminée automatiquement par JSF en fonction des informations contenues dans la propriété Accept-Language de l'en-tête de la requête et du fichier de configuration.



La langue peut aussi être forcée dans l'objet de type view en précisant le code langue dans l'attribut locale.

Exemple :

```
...
          <f:view locale="en">
...
```



Elle peut aussi être déterminée dans le code des traitements. L'exemple suivant va permettre à l'utilisateur de sélectionner la langue utilisée entre français et anglais grâce à deux petites icônes cliquables.

Exemple :

```
<html>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<f:view>
<f:loadBundle basename="com.jmd.test.jsf.Messages" var="msg"/>
<head>
<title>Application de tests avec JSF</title>
</head>
<body>
  <h:form>

  <table>
    <tr>
      <td>
        <h:commandLink action="#{langueApp.activerFR}" immediate="true">
```

```
          <h:graphicImage value="images/francais.jpg" style="border: 0px"/>
        </h:commandLink>
      </td>
      <td>
        <h:commandLink action="#{langueApp.activerEN}" immediate="true">
          <h:graphicImage value="images/anglais.jpg" style="border: 0px"/>
        </h:commandLink>
      </td>
      <td width="100%"> </td>
    </tr>
  </table>

  <h3><h:outputText value="#{msg.login_titre}" /></h3>
  <p> </p>
  <h:panelGrid columns="2">
    <h:outputText value="#{msg.login_nom}" />
    <h:panelGroup>
      <h:inputText value="#{login.nom}" id="nom" required="true"
        binding="#{login.inputTextNom}"/>
      <h:message for="nom"/>
    </h:panelGroup>
    <h:outputText value="#{msg.login_mdp}" />
    <h:inputSecret value="#{login.mdp}"/>
    <h:commandButton value="#{msg.login_valider}" action="login"/>
  </h:panelGrid>

  </h:form>
  </body>
</f:view>
</html>
```

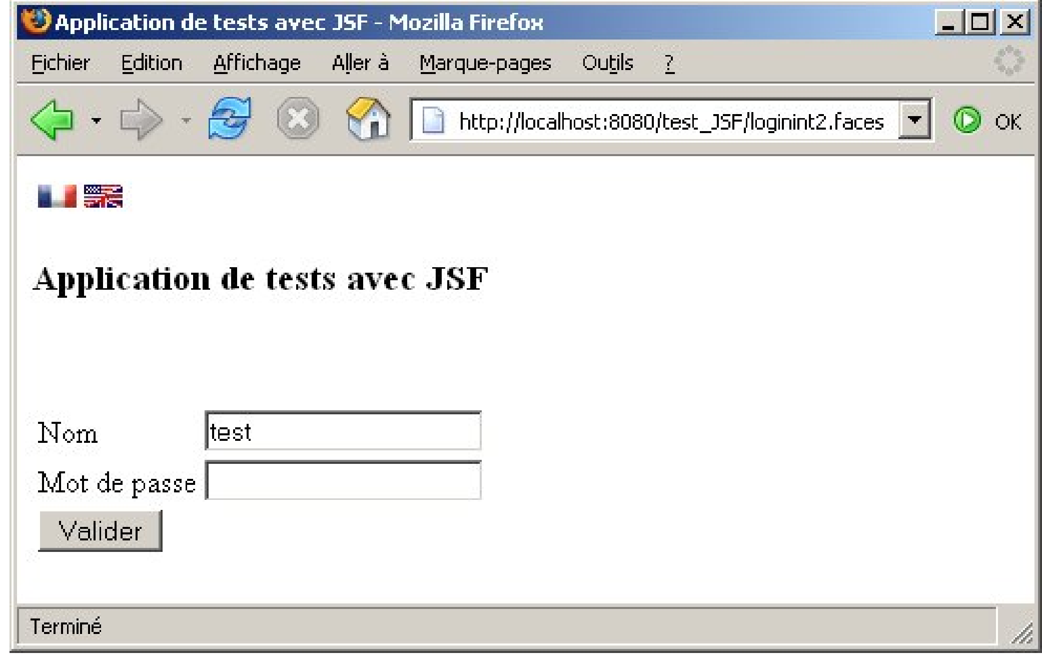
Ce code n'a rien de particulier si ce n'est l'utilisation de l'attribut immediate sur les liens sur le choix de la langue pour empêcher la validation des données lors d'un changement de la langue d'affichage.

Ce sont les deux méthodes du bean qui ce charge de modifier la Locale par défaut du contexte de l'application

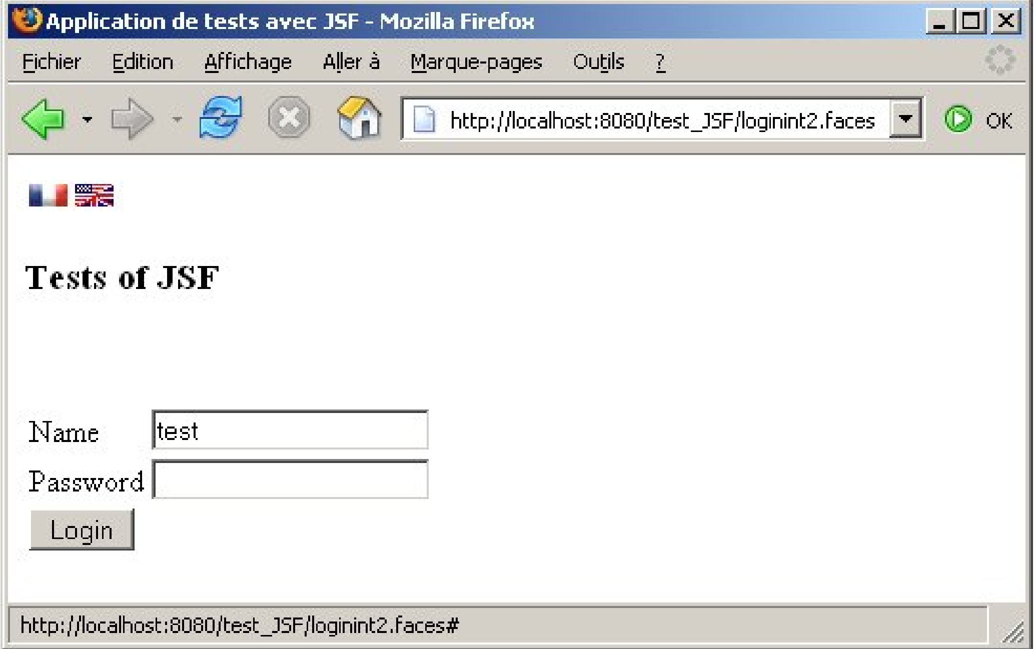
Exemple :

```
package com.jmd.test.jsf;

import java.util.Locale;

import javax.faces.context.FacesContext;

public class LangueApp {

  public String activerFR() {
    FacesContext context = FacesContext.getCurrentInstance();
    context.getViewRoot().setLocale(Locale.FRENCH);
    return null;
  }

  public String activerEN() {
    FacesContext context = FacesContext.getCurrentInstance();
    context.getViewRoot().setLocale(Locale.ENGLISH);
    return null;
  }

}
```

Lors de l'exécution, la page s'affiche en français par défaut.



Lors d'un clic sur la petite icône indiquant la langue anglaise, la page est réaffichée en anglais.



59.19. Les points faibles de JSF

Malgré ces nombreux points forts, JSF possède aussi quelques points faibles :

- Maturité de la technologie

  JSF est une technologie récente qui nécessite l'écriture de beaucoup de code. Bien que prévu pour être utilisé dans des outils pour faciliter la rédaction d'une majeure partie de ce code, seuls quelques outils supportent JSF.

- Manque de composants évolués en standard

  L'implémentation standard ne propose que des composants simples dont la plupart ont une correspondance directe en HTML. Hormis le composant dataTable aucun composant évolué n'est proposé en standard dans la

version 1.0. Il est donc nécessaire de développer ces propres composants ou d'acquérir les composants nécessaires auprès de tiers.

- Consommation en ressources d'une application JSF

  L'exécution d'une application JSF est assez gourmande en ressource notamment mémoire à cause du mode de fonctionnement du cycle de traitement d'une page. Ce cycle de vie inclus la création en mémoire d'une arborescence des composants de la page utilisée lors des différentes étapes de traitements.

- Le rendu des composants uniquement en HTML en standard

  Dans l'implémentation de référence le rendu des composants est uniquement possible en HTML, alors que JSF intègre en système de rendu (Renderer) découplé des traitements des composants. Pour un rendu différent de HTML, il est nécessaire développer ces propres Renderers ou d'acquérir un système de rendu auprès de tiers

La suite de ce chapitre sera développée dans une version future de ce document

60. D'autres frameworks pour les applications web

Ce chapitre sera développé dans une version future de ce document

60.1. stxx

stxx est un projet open source dont le but est de fournir une extension de Struts qui facilite l'utilisation de XML et de XSL en intégration dans Struts.

Le site officiel de ce projet est à l'url : http://stxx.sourceforge.net/.

60.2. WebMacro

Webmacro est un moteur de template open source.

Le site officiel de Webmacro est à l'url : http://www.webmacro.org/

60.3. FreeMarker

FreeMarker est un moteur de template open source développé en Java.

Le site officiel de FreeMarker est à l'url : http://freemarker.sourceforge.net/

60.4. Velocity

Velocity est un moteur de template open source développé en Java par le projet Jakarta du groupe Apache.

http://jakarta.apache.org/velocity/

Cette partie est consacrée aux développements d'applications de type RIA (Rich Internet Application) et RDA (Rich Desktop Application). Ce type d'applications était déjà réalisable respectivement avec les technologies applets et Java Web Start. D'autres technologies standards ou open source sont apparues pour fournir de nouveaux moyens de les développer.

Cette partie contient plusieurs chapitres :

- Les applications riches : RIA et RDA : présente les caractéristiques des applications riches et les principales solutions qui permettent de les développer.
- Les applets : plonge au coeur des premières applications qui ont rendu Java célèbre
- Java Web Start (JWS) : est une technologie qui permet le déploiement d'applications clientes riches à travers le réseau via un navigateur
- Ajax : présente ce concept qui permet de rendre les applications web plus conviviale et plus dynamique. Le framework open source DWR est aussi détaillé.
- GWT (Google Web Toolkit) : GWT est un framework pour le développement d'applications de type RIA

61. Les applications riches : RIA et RDA

Les applications de types client / serveur offrent une bonne ergonomie pour les utilisateurs mais possèdent de nombreux inconvénients notamment au niveau de la maintenance et surtout du déploiement.

Pour palier à ces inconvénients, les applications web se sont répandues. Elles reposent sur des traitements métier côté serveur et une IHM sur un client léger utilisant un simple navigateur web. Malheureusement, ce type d'application ne satisfait pas les utilisateurs notamment car elles offrent des régressions au niveau de l'ergonomie et des interactions.

Les applications riches tentent de réconcilier les avantages des applications C/S et web en conservant le meilleur des deux types d'applications : facilité de déploiement, ergonomie et expérience utilisateur enrichie.

Le développement d'applications web avec Java met généralement en oeuvre un framework reposant sur le modèle MVC tel que Struts ou Spring MVC qui génère sur le serveur des pages HTML retournées au navigateur de l'utilisateur.

Généralement ces frameworks imposent une requête http vers le serveur qui regénèrent toute la page pour tenir compte des modifications ou redirigent vers une autre page. Ceci impose des limitations dans les possibilités offertes par les applications en terme d'expérience utilisateur.

Ces limitations sont influencées par la capacité des navigateurs :

- Non support complet ni homogène des standards (HTML, CSS, ...)
- Incompatibilité de Javascript entre les différents navigateurs
- Certains composants graphiques nécessitent parfois d'être réécrits (onglets, pagination de données, wizard, treeview, ...)
- La sauvegarde de l'état d'une application repose généralement sur les cookies
- ...

Les applications de type RIA proposent une solution pour fournir aux applications exécutées dans un navigateur une expérience utilisateur proche des celles des applications standalone en proposant des fonctionnalités étendues notamment :

- Des composants graphiques évolués sont proposés (barre de menu, onglets, treeview, grille de données, ...)
- Support du drag and drop
- Support multi navigateur avec le même code
- Une meilleure réactivité grâce à un rafraichissement partiel de la page par des appels serveurs via des requêtes http pour obtenir uniquement les données à modifier dans la page. Le format utilisé peut varier selon les solutions utilisées : XML, JSON, ...
- Maintient de l'état de l'application côté client
- Un enrichissement des fonctionnalités graphiques notamment via des effets visuels et une intégration forte du multimédia
- ...

Les applications riches peuvent être regroupées dans deux grandes catégories :

- RIA : Rich Internet Applications
- RDA : Rich Desktop Applications

61.1. Les applications de type RIA

Les applications de type RIA utilisent un navigateur pour la partie IHM de l'application. Pour permettre d'améliorer l'expérience utilisateur des applications, elles utilisent des technologies existantes depuis longtemps mais partiellement ou pas du tout exploité. C'est notamment le cas de la technologie AJAX (Asynchronous JavaScript And Xml).

Il y a plusieurs solutions pour mettre en oeuvre Ajax :

- Tout développer manuellement en utilisant Javascript et DHTML
- Utiliser des bibliothèques de composants tels que Prototype, Script.aculo.us, Dojo, Yahoo ! UI, Rico, Rialto, Ext, jQuery, ...
- Utiliser des frameworks tel que DWR

Les applications RIA peuvent utiliser uniquement les possibilités du navigateur ou avoir besoin d'un plug-in qui fournit un environnement d'exécution.

Les RIA ont cependant un certain nombre d'inconvénients :

- La multitude des solutions proposées et leur immaturité
- Les utilisateurs doivent adapter leur mode de navigation
- L'accessibilité est rarement assurée d'autant que ces solutions sont très riches
- Le référencement est parfois difficile
- ...

Les solutions RIA proposent généralement un environnement d'exécution, des bibliothèques et/ou des API, et des outils qui permettent d'être plus efficace et plus riche que le simple ajout d'Ajax dans une application de façon manuelle.

61.2. Les applications de type RDA

Les applications de type RDA reposent sur les technologies des applications de type web mais elles s'exécutent sur le bureau donc sans navigateur web. Elles permettent d'avoir les mêmes fonctionnalités qu'une application de type RIA mais exécutées hors du navigateur.

Elles nécessitent un environnement d'exécution installé sur le poste client généralement sous la forme d'une machine virtuelle avec un ensemble d'API.

Elles offrent de meilleures interactivités notamment avec le système sous jacent (drag & drop, accès au système de fichiers, ...). Les applications de type RDA peuvent avoir un accès au système sous jacent sous réserve d'être signées pour des raisons de sécurité. Cela permet un meilleure interactivité avec le système pour par exemple permettre une utilisation en mode déconnectée de l'application.

De plus, ces applications permettent généralement de se télécharger sur internet et se mettre à jour via le réseau.

61.3. Les contraintes

Le développement d'applications de type RIA doit tenir compte de certaines contraintes inhérentes à ce type d'applications.

Les développeurs doivent utiliser les solutions RIA dans la limite de ce qu'elles peuvent proposer : toutes les applications ne peuvent pas être de type RIA. Par exemple, les applications de type RIA ne sont généralement pas adaptées pour de grandes applications manipulant de grande quantités de données.

Les développeurs d'applications web traditionnelles doivent tenir compte du mode de mise en oeuvre des applications RIA : la conception doit tenir compte du fait que l'application ne fonctionne pas sur un mode de rafraichissement à chaque requête/réponse. Une application RIA est ainsi responsable du rafraichissement de ses données.

Le développement d'une application de type RIA nécessite la mise en oeuvre d'une architecture notamment côté serveur pour permettre de fournir à l'application les données et les traitements métiers nécessaires. Généralement, les solutions RIA ne concernent que la partie présentation et ne proposent aucune fonctionnalité dédiée pour la partie backend.

Lors de l'évaluation d'une solution, il est nécessaire d'évaluer ses capacités d'intégration avec la partie backend pour permettre les échanges de données et l'invocation de traitements métiers.

Les applications nécessitent plus l'intervention de graphistes pour définir les graphiques requis par l'application.

61.4. Les solutions RIA

Le besoin grandissant du marché concernant les applications riches se reflète dans l'activité des grands acteurs du marché comme Adobe, Sun, Microsoft, Google, ...

Ainsi, de nombreuses solutions sont proposées pour permettre le développement et la mise en oeuvre des applications riches. La plupart de ces solutions sont récentes et sont encore en cours de développement. Ces solutions ne sont donc pas toutes fiables mais elles évoluent très rapidement pour permettre de répondre à la demande importante du marché.

Parmi ces solutions, en plus des solutions reposant sur Java, il y a notamment Adobe Flex/Air et Microsoft Silverlight.

61.4.1. Les solutions RIA reposant sur Java

Dans le monde Java, Sun propose Java FX. La fondation Eclipse propose Eclipse RCP (Rich Client Plateform) pour le développement d'applications de type RDA. Wazaabi repose sur RCP et XUL.

De nombreux frameworks open source facilitent aussi le développement de nouvelles applications de type RIA notamment :

- GWT (Google Web Toolkit)
- ZK
- Echo
- Ice Faces, Rich Faces
- Wicket
- TIBCO General Interface
- ...

61.4.1.1. Sun Java FX

Java FX est un ensemble de technologies proposé par Sun pour le développement d'applications de type RIA.

Java FX a été présenté pour la première fois au JavaOne 2007 et a été la technologie mise en avant lors du JavaOne 2008.

Elle est arrivée tardivement notamment vis-à-vis de Flex et est de plus restée une bonne année sans réel outils : un interpréteur était disponible mais aucun compilateur ni IDE.

Depuis 2008, Java FX s'est enrichi d'outils notamment un SDK et de fonctionnalités multimédia avancées grâce à l'intégration de codecs audio et vidéo.

Le grand intérêt de Java FX est son intégration avec Java. Par défaut, il faut coder l'application en utilisant Java FX Script qui est un langage de scripting déclaratif.

L'avantage de Java FX est qu'il nécessite une machine virtuelle Java (JVM) de la plate-forme SE ou ME pour s'exécuter : cela apporte à Java FX un avantage certain car une JVM est installée sur une très large majorité d'appareils de différent type : notamment sur les ordinateurs de bureau et portable, encore plus sur les appareils téléphoniques mobiles et présent dans tous les lecteurs de disque Blu-Ray.

Une caractéristique de Java FX est d'être intégralement open source, ce qui n'est pas entièrement le cas de ces principaux concurrents directs.

Sun travaille sur plusieurs autres modules de la plate-forme Java FX notamment Java FX Mobile et Java FX TV.

Plusieurs sites relatifs à Java FX peuvent être consultés :

- JavaFX
- http://java.sun.com/javafx/
- Open JavaFX

61.4.1.2. Google GWT

Google propose GWT (Google Web Toolkit) pour le développement d'applications de type RIA.

L'application est écrite en Java avec un sous ensemble de l'API standard et une API dédiée proposée par Google. L'ensemble du code est compilé pour générer du code Javascript optimisé pour chaque navigateur.

Hormis une page hôte et l'utilisation des feuilles de style CSS, le développeur n'a besoin d'aucune connaissance sur les technologies web car elles sont encapsulées dans l'API. Le code Java écrit avec GWT ressemble plus à celui utilisé pour des applications AWT qu'une application de type web.

Les composants graphiques proposés par GWT sont relativement basiques mais des bibliothèques tierces permettent de fournir des composants évolués notamment grâce à la facilité d'encapsuler du code Javascript dans GWT.

Le site officiel est à l'url http://code.google.com/webtoolkit/

61.4.1.3. ZK

Le framework ZK est un framework open source pour le développement d'applications de type RIA mettant en oeuvre Ajax.

Pour le développement de l'interface graphique, ZK propose XUML (ZK User Interface Markup Langage) qui permet une description de l'interface en XML grâce à des composants XUL et XHTML.

ZK propose une gestion des événements et une intégration avec d'autres frameworks Java

Plusieurs langages sont supportés pour coder les traitements dont le principal est Java.

Le site officiel est à l'url http://www.zkoss.org/

61.4.1.4. Echo

Echo est un framework open source pour le développement orienté objet avec gestion des événements d'applications web riches.

Le développement de la partie IHM ressemble au développement d'applications graphique de type client lourd : composants orientés objets, gestion des événements, ...

Selon la version du framework Echo utilisé, une application peut prendre deux formes :

- Entièrement orientée serveur (Echo 2 et 3)
- Avoir une partie en Javascript côté client (Echo 3 uniquement).

Le site officiel est à l'url http://echo.nextapp.com/site/

61.4.1.5. Apache Wicket

Wicket est un projet de la fondation Apache qui propose un framework orienté composants pour le développement d'applications web riches.

Le framework propose une séparation entre la partie présentation en XHTML et la partie traitement écrite en Java via des composants.

La page est encapsulée dans un objet et représentée dans une page XHTML dans lequel on ajoute des composants graphiques. La liaison se fait par un id.

Le site officiel est à l'url http://wicket.apache.org/

61.4.1.6. Les composants JSF

Plusieurs composants JSF proposent une implémentation d'Ajax dans leurs composants notamment :

- Myfaces Tobago :
- Myfaces Trinidad :
- IceFaces :
- Jboss RichFaces :
- ...

61.4.1.7. Tibco General Interface

General Interface est un framework open source pour le développement d'applications web riches. General Interface est diffusée en open source sous la licence BSD et sous une forme commerciale avec un support.

General Interface propose un IDE qui facilite le développement de la partie graphique d'une application en proposant d'utiliser le cliquer/glisser des composants.

Les échanges entre le client et le serveur se font via des services web : ceci permet de rendre le framework GI plus indépendant de la solution backend utilisée.

L'application peut être exécutée dans Internet Explorer et Firefox sous Windows, Linux et Mac.

Le site officiel est à l'url http://gi.tibco.com/

61.4.1.8. Eclipse RAP

Eclipse RAP (Rich Ajax Platform) repose sur l'API Eclipse RCP et génère une application html utilisant Ajax.

61.4.2. Les autres solutions RIA

Plusieurs fournisseurs proposent des solutions pour le développements d'applications de type RIA. Généralement ces solutions se concentrent sur la partie IHM et s'interface plus ou moins facilement avec un backend écrit en Java notamment au travers de services web par exemple.

61.4.2.1. Adobe Flex

Adobe Flex est un outil de développement pour créer des applications compilé sous la forme de fichier swf exécuté dans le Flash player.

Adobe Flex repose sur MXML qui permet de créer l'interface graphique de manière déclarative en XML et ActionScript pour être compilés en une application Flash nécessitant le plug-in Flash pour être exécutée dans un navigateur.

Le site officiel d'Adobe Flex est à l'url : http://labs.adobe.com/technologies/flex

Flex se concentre sur la partie IHM et permet une intégration facilité avec un backend développé en Java avec notamment une solution open source : Blaze DS.

Le site http://flex.org/ propose de nombreuses ressources pour Flex.

61.4.2.2. Microsoft Silverlight

Silverlight (initialement connu sous le nom WPF/e) est la solution proposée par Microsoft pour le développement d'applications de type RIA.

Microsoft Silverlight repose sur XAML qui permet de décrire l'interface graphique en XML. Le plug-in Silverlight est requis pour l'exécution d'une application.

La version 1.0 utilise le langage Javascript.

La version 2.0 de Silverlight permet de développer des applications avec les langages de la plate-forme .Net. Le plug-in de Silverlight 2.0 incorpore une machine virtuelle de type CLR mais seul un sous ensemble d'API de la plate-forme .Net est utilisable.

Silverlight propose la technologie DeepZone qui permet de faire des zooms sur une image.

Le site officiel de Microsoft Silverlight est à l'url : http://www.silverlight.net

Le site Web officiel du framework Microsoft .NET 3 : http://msdn2.microsoft.com/fr-fr/netframework

61.4.2.3. Google Gears

Google gears est une Api et un plug-in qui permet d'utiliser une base de données SQLite pour stocker des données en local et ainsi permettre à des applications Ajax de fonctionner en mode déconnecté. Un exemple de mise en oeuvre de cette API est proposé par Google Reader.

61.4.3. Une comparaison entre GWT et Flex

Le but est de fournir les principaux avantages et inconvénients de GWT et Flex.

	Avantages	Inconvénients
GWT	Nécessite uniquement un navigateur (pas de plug-in) Pas de nouveau langage à apprendre pour un développeur Java	Peu de composants graphiques évolués qui nécessitent généralement l'utilisation d'une bibliothèque tierce Pas de structure standard (type MVC) pour les applications

Flex	Richesse et cohérence des composants graphiques Rendu identique sur les navigateurs supportés	Nouveaux langages (MXML et ActionScript) à apprendre pour les développeurs Java, en plus de Java nécessaires pour développer la partie serveur Nécessite la plug-in flash (il est cependant très largement déployé)

GWT et Flex possèdent des avantages et inconvénients communs :

- Les outils pour être productifs sont payant avec les deux frameworks : par exemple, Adobe propose un plug-in Eclipse (Flex Builder)
- Leur communauté est importante et très productive
- Pour des sites web, le référencement est délicat avec les deux solutions

61.5. Les solutions RDA

Les solutions pour développer des applications de type RDA existent déjà sous plusieurs formes :

- Java avec Java Web Start
- Java avec des socles applicatifs : Eclipse RCP ou Netbeans RCP
- Adobe AIR

61.5.1. Adobe AIR

Adobe AIR (Adobe Integrated Runtime) propose un environnement d'exécution pour application Flex et/ou Html/Javascript.

Le site officiel d'Adobe AIR est à l'url : http://labs.adobe.com/technologies/air

La suite de cette section sera développée dans une version future de ce document

61.5.2. Eclipse RCP

Eclipse RCP (Rich Client Platform) est la base sur laquelle Eclipse repose. Ce socle utilise Java et SWT.

La suite de cette section sera développée dans une version future de ce document

61.5.3. Netbeans RCP

NetBeans RCP est la base sur laquellz NetBeans repose. Ce socle utilise Java et Swing.

La suite de cette section sera développée dans une version future de ce document

62. Les applets

Une applet est un programme Java qui s'exécute dans un logiciel de navigation supportant java ou dans l'appletviewer du JDK.

Attention : il est recommandé de tester les applets avec l'appletviewer car les navigateurs peuvent prendre l'applet contenu dans leur cache plutôt que la dernière version compilée.

Le mécanisme d'initialisation d'une applet se fait en deux temps :

1. la machine virtuelle Java instancie l'objet Applet en utilisant le constructeur par défaut
2. la machine virtuelle Java envoie le message init() à l'objet Applet

Ce chapitre contient plusieurs sections :

- L'intégration d'applets dans une page HTML
- Les méthodes des applets
- Les interfaces utiles pour les applets
- La transmission de paramètres à une applet
- Les applets et le multimédia
- Un applet pouvant s'exécuter comme une application
- Les droits des applets

62.1. L'intégration d'applets dans une page HTML

Dans une page HTML, il faut utiliser le tag APPLET avec la syntaxe suivante :

<APPLET CODE=« Exemple.class » WIDTH=200 HEIGHT=300 > </APPLET>

Le nom de l'applet est indiqué entre guillemets à la suite du paramètre CODE.

Les paramètres WIDTH et HEIGHT fixent la taille de la fenêtre de l'applet dans la page HTML. L'unité est le pixel. Il est préférable de ne pas dépasser 640 * 480 (VGA standard).

Le tag APPLET peut comporter les attributs facultatifs suivants :

Tag	Rôle
CODEBASE	permet de spécifier le chemin relatif par rapport au dossier de la page contenant l'applet. Ce paramètre suit le paramètre CODE. Exemple : CODE=nomApplet.class CODEBASE=/nomDossier
HSPACE et VSPACE	permettent de fixer la distance en pixels entre l'applet et le texte
ALT	

	affiche le texte spécifié par le paramètre lorsque le navigateur ne supporte pas Java ou que son support est désactivé.

Le tag PARAM permet de passer des paramètres à l'applet. Il doit être inclus entre les tags APPLET et /APPLET.

<PARAM nomParametre value=« valeurParametre »> </APPLET>

La valeur est toujours passée sous forme de chaîne de caractères donc entourée de guillemets.

Exemple : <APPLET code=« Exemple.class » width=200 height=300>

Le texte contenu entre <APPLET> et </APPLET> est afficher si le navigateur ne supporte pas java.

62.2. Les méthodes des applets

Une classe dérivée de la classe java.applet.Applet hérite de méthodes qu'il faut redéfinir en fonction des besoins et doit être déclarée public pour fonctionner.

En général, il n'est pas nécessaire de faire un appel explicite aux méthodes init(), start(), stop() et destroy() : le navigateur se charge d'appeler ces méthodes en fonction de l'état de la page HTML contenant l'applet.

62.2.1. La méthode init()

Cette méthode permet l'initialisation de l'applet : elle n'est exécutée qu'une seule et unique fois après le chargement de l'applet.

62.2.2. La méthode start()

Cette méthode est appelée automatiquement après le chargement et l'initialisation (via la méthode init()) lors du premier affichage de l'applet.

62.2.3. La méthode stop()

Le navigateur appelle automatiquement la méthode lorsque l'on quitte la page HTML. Elle interrompt les traitements de tous les processus en cours.

62.2.4. La méthode destroy()

Elle est appelée après l'arrêt de l'applet ou lors de l'arrêt de la machine virtuelle. Elle libère les ressources et détruit les threads restants

62.2.5. La méthode update()

Elle est appelée à chaque rafraichissement de l'écran ou appel de la méthode repaint(). Elle efface l'écran et appelle la méthode paint(). Ces actions provoquent souvent des scintillements. Il est préférable de redéfinir cette méthode pour qu'elle n'efface plus l'écran :

Exemple :

```
public void update(Graphics g) { paint (g);}
```

62.2.6. La méthode paint()

Cette méthode permet d'afficher le contenu de l'applet à l'écran. Ce rafraîchissement peut être provoqué par le navigateur ou par le système d'exploitation si l'ordre des fenêtres ou leur taille ont été modifiés ou si une fenêtre recouvre l'applet.

Exemple :

```
public void paint(Graphics g)
```

La méthode repaint() force l'utilisation de la méthode paint().

Il existe des méthodes dédiées à la gestion de la couleur de fond et de premier plan

La méthode setBackground(Color), héritée de Component, permet de définir la couleur de fond d'une applet. Elle attend en paramètre un objet de la classe Color.

La méthode setForeground(Color) fixe la couleur d'affichage par défaut. Elle s'applique au texte et aux graphiques.

Les couleurs peuvent être spécifiées de trois manières différentes :

utiliser les noms standards prédéfinis	Color.nomDeLaCouleur Les noms prédéfinis de la classe Color sont : black, blue, cyan, darkGray, gray, green, lightGray, magenta, orange, pink, red, white, yellow

utiliser 3 nombres de type entier représentant le RGB	(Red,Green,Blue : rouge,vert, bleu)
	
	Exemple :
	`Color macouleur = new Color(150,200,250);`
	`setBackground (macouleur);`
	`// ou setBackground(150,200,250);`

utiliser 3 nombres de type float utilisant le système HSB	(Hue, Saturation, Brightness : teinte, saturation, luminance). Ce système est moins répandu que le RGB mais il permet notamment de modifier la luminance sans modifier les autres caractéristiques Exemple : `setBackground(0.0,0.5,1.0);` dans ce cas 0.0,0.0,0.0 représente le noir et 1.0,1.0,1.0 représente le blanc.

62.2.7. Les méthodes size() et getSize()

L'origine des coordonnées en Java est le coin supérieur gauche. Elles s'expriment en pixels avec le type int.

La détermination des dimensions d'une applet se fait de la façon suivante :

Exemple (code Java 1.0) :

```
Dimension dim = size();
int applargeur = dim.width;
int apphauteur = dim.height;
```

Avec le JDK 1.1, il faut utiliser getSize() à la place de size().

Exemple (code Java 1.1) :

```
public void paint(Graphics g) {
   super.paint(g);
   Dimension dim = getSize();
   int applargeur = dim.width;
   int apphauteur = dim.height;
   g.drawString("width = "+applargeur,10,15);
   g.drawString("height = "+apphauteur,10,30);
}
```

62.2.8. Les méthodes getCodeBase() et getDocumentBase()

Ces méthodes renvoient respectivement l'emplacement de l'applet sous forme d'adresse Web ou de dossier et l'emplacement de la page HTML qui contient l'applet.

Exemple :

```
public void paint(Graphics g) {
   super.paint(g);
   g.drawString("CodeBase = "+getCodeBase(),10,15);
   g.drawString("DocumentBase = "+getDocumentBase(),10,30);
}
```

62.2.9. La méthode showStatus()

Affiche un message dans la barre de statut de l'applet

Exemple :

```
public void paint(Graphics g) {
   super.paint(g);
   showStatus("message à afficher dans la barre d'état");
}
```

62.2.10. La méthode getAppletInfo()

Permet de fournir des informations concernant l'auteur, la version et le copyright de l'applet

Exemple :

```
static final String appletInfo = " test applet : auteur, 1999 \n\nCommentaires";

public String getAppletInfo() {
   return appletInfo;
}
```

Pour voir les informations, il faut utiliser l'option info du menu Applet de l'appletviewer.

62.2.11. La méthode getParameterInfo()

Cette méthode permet de fournir des informations sur les paramètres reconnus par l'applet

Le format du tableau est le suivant :

{ {nom du paramètre, valeurs possibles, description} , ... }

Exemple :

```
static final String[][] parameterInfo =
{ {"texte1", "texte1", " commentaires du texte 1" } ,
  {"texte2", "texte2", " commentaires du texte 2" } };

public String[][] getParameterInfo() {
   return parameterInfo;
}
```

Pour voir les informations, il faut utiliser l'option info du menu Applet de l'appletviewer.

62.2.12. La méthode getGraphics()

Elle retourne la zone graphique d'une applet : utile pour dessiner dans l'applet avec des méthodes qui ne possèdent pas le contexte graphique en paramètres (ex : mouseDown ou mouseDrag).

62.2.13. La méthode getAppletContext()

Cette méthode permet l'accès à des fonctionnalités du navigateur.

62.2.14. La méthode setStub()

Cette méthode permet d'attacher l'applet au navigateur.

62.3. Les interfaces utiles pour les applets

62.3.1. L'interface Runnable

Cette interface fournit le comportement nécessaire à un applet pour devenir un thread.

Les méthodes start() et stop() de l'applet peuvent permettre d'arrêter et de démarrer un thread pour permettre de limiter l'usage des ressources machines lorsque la page contenant l'applet est inactive.

62.3.2. L'interface ActionListener

Cette interface permet à l'applet de répondre aux actions de l'utilisateur avec la souris

La méthode actionPerformed() permet de définir les traitements associés aux événements.

Exemple (code Java 1.1) :

```
public void actionPerformed(ActionEvent evt) { ... }
```

Pour plus d'information, voir le chapitre «L'interception des actions de l'utilisateur».

62.3.3. L'interface MouseListener pour répondre à un clic de souris

Exemple (code Java 1.1) :

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class AppletMouse extends Applet implements MouseListener {
   int nbClick = 0;

   public void init() {
      super.init();
      addMouseListener(this);
   }

   public void mouseClicked(MouseEvent e) {
      nbClick++;
      repaint();
   }

   public void mouseEntered(MouseEvent e) {
   }

   public void mouseExited(MouseEvent e) {
   }

   public void mousePressed(MouseEvent e) {
   }

   public void mouseReleased(MouseEvent e) {
   }

   public void paint(Graphics g) {
      super.paint(g);
      g.drawString("Nombre de clics : " + nbClick, 10, 10);
   }

}
```

Pour plus d'information, voir le chapitre sur «L'interception des actions de l'utilisateur».

62.4. La transmission de paramètres à une applet

La méthode getParameter() retourne les paramètres écrits dans la page HTML. Elle retourne une chaîne de caractères de type String.

Exemple :

```
Sting parametre;
parametre = getParameter(" nom-parametre ");
```

Si le paramètre n'est pas renseigné dans la page HTML alors getParameter() retourne null

Pour utiliser les valeurs des paramètres, il sera souvent nécessaire de faire une conversion de la chaîne de caractères dans le type voulu en utilisant les Wrappers

Exemple :

```
String taille;
int hauteur;
taille = getParameter(" hauteur ");
Integer temp = new Integer(taille)
hauteur = temp.intValue();
```

Exemple :

```
int vitesse;
String paramvitesse = getParameter(" VITESSE ");
if (paramvitesse != null) vitesse = Integer.parseInt(paramVitesse);
// parseInt ne fonctionne pas avec une chaîne vide
```

Attention : l'appel à la méthode getParameter() dans le constructeur pas défaut lève une exception de type NullPointerException.

Exemple :

```
public MonApplet() {
   String taille;
   taille = getParameter(" message ");
}
```

62.5. Les applets et le multimédia

62.5.1. L'insertion d'images

Java supporte deux standards :

- le format GIF de Compuserve qui est beaucoup utilisé sur internet car il génère des fichiers de petite taille contenant des images d'au plus 256 couleurs.
- et le format JPEG. qui convient mieux aux grandes images et à celles de plus de 256 couleurs car le taux de compression avec perte de qualité peut être précisé.

Pour la manipulation des images, le package nécessaire est java.awt.image.

La méthode getImage() possède deux signatures : getImage(URL url) et getImage (URL url, String name).

On procède en deux étapes : le chargement puis l'affichage. Si les paramètres fournis à getImage ne désignent pas une image, aucune exception n'est levée.

La méthode getImage() ne charge pas de données sur le poste client. Celles ci seront chargées quand l'image sera dessinée pour la première fois.

Exemple :

```
public void paint(Graphics g) {
   super.paint(g);
   Image image=null;
   image=getImage(getDocumentBase( ), "monimage.gif"); //chargement de l'image
   g.drawImage(image, 40, 70, this);
}
```

Le sixième paramètre de la méthode drawImage() est un objet qui implémente l'interface ImageObserver. ImageObserver est une interface déclarée dans le package java.awt.image qui sert à donner des informations sur le fichier image. Souvent, on indique this à la place de cet argument représentant l'applet elle même. La classe ImageObserver détecte le chargement et la fin de l'affichage d'une image. La classe Applet contient le comportement qui se charge de faire ces actions d'ou le fait de mettre this.

Pour obtenir les dimensions de l'image à afficher on peut utiliser les méthodes getWidth() et getHeight() qui retourne un nombre entier en pixels.

Exemple :

```
int largeur = 0;
int hauteur = 0;
largeur = image.getWidth(this);
hauteur = image.getHeight(this);
```

62.5.2. L'utilisation des capacités audio

Seul le format d'extension .AU de Sun est supporté par java. Pour utiliser un autre format, il faut le convertir.

La méthode play() permet de jouer un son.

Exemple :

```
import java.net.URL;

   ...
   try {
      play(new URL(getDocumentBase(), " monson.au "));
   } catch (java.net.MalformedURLException e) {}
```

La méthode getDocumentBase() détermine et renvoie l'URL de l'applet.

Ce mode d'exécution n'est valable que si le son n'est à reproduire qu'une seule fois, sinon il faut utiliser l'interface AudioClip.

Avec trois méthodes, l'interface AudioClip facilite l'utilisation des sons :

- public abstract void play() // jouer une seule fois le fichier
- public abstract void loop() // relancer le son jusqu'à interruption par la méthode stop ou la fin de l'applet
- public abstract void stop() // fin de la reproduction du clip audio

Exemple :

```
import java.applet.*;
import java.awt.*;
import java.net.*;

public class AppletMusic extends Applet {
   protected AudioClip aC = null;
```

```
   public void init() {
      super.init();
      try {
      AppletContext ac = getAppletContext();
      if (ac != null)
         aC = ac.getAudioClip(new URL(getDocumentBase(), "spacemusic.au"));
      else
         System.out.println(" fichier son introuvable ");
      }
      catch (MalformedURLException e) {}
      aC.loop();
   }
}
```

Pour utiliser plusieurs sons dans une applet, il suffit de déclarer plusieurs variables AudioClip.

L'objet retourné par la méthode getAudioClip() est un objet qui implémente l'interface AudioClip défini dans la machine virtuelle car il est très dépendant du système de la plate forme d'exécution.

62.5.3. L'animation d'un logo

Exemple :

```
import java.applet.*;
import java.awt.*;

public class AppletAnimation extends Applet implements Runnable {
   Thread thread;
   protected Image tabImage[];
   protected int index;

   public void init() {
      super.init();
      //chargement du tableau d'image
      index = 0;
      tabImage = new Image[2];
      for (int i = 0; i < tabImage.length; i++) {
         String fichier = new String("monimage" + (i + 1) + ".gif ");
         tabImage[i] = getImage(getDocumentBase(), fichier);
      }
   }

   public void paint(Graphics g) {
      super.paint(g);
      // affichage de l'image
      g.drawImage(tabImage[index], 10, 10, this);
   }

   public void run() {
      //traitements exécuté par le thread
      while (true) {
         repaint();
         index++;
         if (index >= tabImage.length)
            index = 0;
         try {
            thread.sleep(500);
         }
         catch (InterruptedException e) {
            e.printStackTrace();
         }
      }
   }

   public void start() {
      //demarrage du tread
      if (thread == null) {
         thread = new Thread(this);
```

```
            thread.start();
        }
    }

    public void stop() {
        // arret du thread
        if (thread != null) {
            thread.stop();
            thread = null;
        }
    }

    public void update(Graphics g) {
        //la redéfinition de la méthode permet d'éviter les scintillements
        paint(g);
    }

}
```

La surcharge de la méthode paint() permet d'éviter le scintillement de l'écran du à l'effacement de l'écran et à son rafraichissement. Dans ce cas, seul le rafraichissement est effectué.

62.6. Un applet pouvant s'exécuter comme une application

Il faut rajouter une classe main à l'applet, définir une fenêtre qui recevra l'affichage de l'applet, appeler les méthodes init() et start() et afficher la fenêtre.

Exemple (code Java 1.1) :

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class AppletApplication extends Applet implements WindowListener {

    public static void main(java.lang.String[] args) {
        AppletApplication applet = new AppletApplication();
        Frame frame = new Frame("Applet");
        frame.addWindowListener(applet);
        frame.add("Center", applet);
        frame.setSize(350, 250);
        frame.show();
        applet.init();
        applet.start();
    }

    public void paint(Graphics g) {
        super.paint(g);
        g.drawString("Bonjour", 10, 10);
    }

    public void windowActivated(WindowEvent e) { }

    public void windowClosed(WindowEvent e) { }

    public void windowClosing(WindowEvent e) {
        System.exit(0);
    }

    public void windowDeactivated(WindowEvent e) { }

    public void windowDeiconified(WindowEvent e) { }

    public void windowIconified(WindowEvent e) { }

    public void windowOpened(WindowEvent e) { }

```

```
}
```

62.7. Les droits des applets

Une applet est une application Java hébergée sur une machine distante (un serveur Web) et qui s'exécute, après chargement, sur la machine client équipée d'un navigateur. Ce navigateur contrôle les accès de l'applet aux ressources locales et ne les autorisent pas systématiquement : chaque navigateur définit sa propre règle.

Le modèle classique de sécurité pour l'exécution des applets, recommandé par Sun, distingue deux types d'applets : les applets non dignes de confiance (untrusted) qui n'ont pas accès aux ressources locales et externes, les applets dignes de confiance (trusted) qui ont l'accès. Dans ce modèle, une applet est par défaut untrusted.

La signature d'une applet permet de désigner son auteur et de garantir que le code chargé par le client est bien celui demandé au serveur. Cependant, une applet signée n'est pas forcement digne de confiance.

La suite de ce chapitre sera développée dans une version future de ce document

63. Java Web Start (JWS)

Java Web Start est une technologie pour permettre le déploiement d'application standalone à travers le réseau, développée avec la plate forme Java 2. Il permet l'installation d'une application grâce à un simple clic dans un navigateur. JWS a été inclus dans le J2RE 1.4. Pour les versions antérieures du J2RE, il est nécessaire de télécharger JWS et de l'installer sur le poste client.

JWS est le résultat des travaux de la JSR-56. La page officielle de Sun concernant cette technologie est : http://java.sun.com/products/javawebstart/

JWS permet la mise à jour automatique de l'application si une nouvelle version est disponible sur le serveur et assure une mise en cache locale des applications pour accélérer leur réutilisation ultérieure.

La sécurité des applications exécutées est assurée par l'utilisation du bac à sable (sandbox) comme pour une applet, dès lors pour certaines opérations il est nécessaire de signer l'application.

JWS utilise et implémente une API et un protocole nommée Java Network Lauching Protocol (JNPL).

Le grand avantage de Java Web Start est qu'il est inutile de modifier une application pour qu'elle puisse être déployée avec cette technologie (à condition que les fichiers contenant des ressources soient accédés en utilisant la méthode getResource() du classloader).

L'application doit être packagée dans un fichier jar qui sera associée sur le serveur à un fichier particulier de lancement.

L'utilisation d'une application via JWS implique la réalisation de plusieurs étapes :

- Packager l'application dans un fichier jar en le signant si nécessaire
- Créer le fichier de lancement .jnlp
- Copier les deux fichiers sur le serveur web

Ce chapitre contient plusieurs sections :

- La création du package de l'application
- La signature d'un fichier jar
- Le fichier JNPL
- La configuration du serveur web
- Le fichier HTML
- Le test de l'application
- L'utilisation du gestionnaire d'applications
- L'API de Java Web Start

63.1. La création du package de l'application

L'application doit être packagée sous la forme d'un fichier .jar.

Il est possible de fournir une petite icône pour représenter l'application : celle si doit avoir une taille de 64 x 64 pixels au format Gif ou JPEG.

63.2. La signature d'un fichier jar

L'exemple de cette section créé un certificat et signe l'application avec ce dernier.

Exemple :

```
C:\java\>keytool -genkey -keystore mes_cles -alias cle_de_test
Tapez le mot de passe du Keystore :  test
Mot de passe de Keystore trop court, il doit compter au moins 6 caractères
Tapez le mot de passe du Keystore :  erreur keytool : java.lang.NullPointerExcep
tion
C:\java\>keytool -genkey -keystore mes_cles -alias cle_de_test
Tapez le mot de passe du Keystore :  mptest
Quels sont vos prénom et nom ?
  [Unknown] :  jean michel
Quel est le nom de votre unité organisationnelle ?
  [Unknown] :  test
Quelle est le nom de votre organisation ?
  [Unknown] :  test
Quel est le nom de votre ville de résidence ?
  [Unknown] :  Metz
Quel est le nom de votre état ou province ?
  [Unknown] :  France
Quel est le code de pays à deux lettres pour cette unité ?
  [Unknown] :  fr
Est-ce CN=jean michel, OU=test, O=test, L=Metz, ST=France, C=fr ?
  [non] :  oui
Spécifiez le mot de passe de la clé pour <cle_de_test>
        (appuyez sur Entrée s'il s'agit du mot de passe du Keystore) :
C:\java\>
C:\java\>keytool -selfcert -alias cle_de_test -keystore mes_cles
Tapez le mot de passe du Keystore :  mptest
C:\java\>keytool -list -keystore mes_cles
Tapez le mot de passe du Keystore :  mptest
Type Keystore : jks
Fournisseur Keystore : SUN
Votre Keystore contient 1 entrÚe(s)
cle_de_test, 12 nov. 2003, keyEntry,
Empreinte du certificat (MD5) : 9E:5A:61:CC:D8:88:02:59:1D:3B:41:C9:CA:26:1D:BD

C:\java>jarsigner -keystore mes_cles -signedjar MonJarSigne.jar MonApp.jar cle_d
e_test
Enter Passphrase for keystore:

Warning:
The signer certificate will expire within six months.

C:\java>dir
 Le volume dans le lecteur C s'appelle SW_Preload
 Le numéro de série du volume est 043F-2ED6

 Répertoire de C:\java

30/08/2009  11:44    <REP>          .
30/08/2009  11:44    <REP>          ..
30/08/2009  11:38             1 259 mes_cles
15/08/2009  23:58            15 793 MonApp.jar
30/08/2009  11:44            17 247 MonJarSigne.jar
               3 fichier(s)           34 299 octets
               0 Rép(s)  70 055 645 184 octets libres
```

63.3. Le fichier JNPL

Ce fichier au format XML permet de décrire l'application.

La racine de ce document XML est composée du tag <jnpl>. Son attribut codebase permet de préciser l'url ou sont stockés les fichiers précisés dans le document via l'attribut href.

Le tag <information> permet de fournir des précisions qui seront utilisées par le gestionnaire d'application sur le poste client. Ce tag possède plusieurs noeuds enfants :

Nom du tag	Rôle
Title	Le nom de l'application
Vendor	Nom de l'auteur de l'application
Homepage	Préciser une page HTML qui contient des informations sur l'application grâce à son attribut href
Description	Une description de l'application. Il est possible de préciser plusieurs types de description grâce à l'attribut kind. Les valeurs possibles sont : one-line, short et tooltip. Pour utiliser plusieurs descriptions, il faut utiliser plusieurs tags Description avec l'attribut kind adéquat
Offline-allowed	Ce tag précise que l'application peut être exécutée dans un mode déconnecté. L'avantage de ne pas préciser ce tag et de s'assurer que la dernière version de l'application est toujours utilisée mais elle nécessite obligatoirement une connexion pour toute exécution.
Icon	Permet de préciser une URL vers une image de 64 x 64 pixels au format gif ou JPEG grâce à l'attribut href

Le tag <security> permet de préciser des informations concernant la sécurité.

Nom du tag	Rôle
All-permissions	Indique que l'application à besoin de tous les droits pour s'exécuter. L'application doit alors être obligatoirement signée. Si ce tag n'est pas précisé alors l'application s'exécute dans le bac à sable et possède les mêmes restrictions qu'une applet au niveau de la sécurité

Le tag <resources> permet de préciser des informations sur les ressources utilisées par l'application. L'attribut os permet de préciser des paramètres pour un système d'exploitation particulier.

Nom du tag	Rôle
J2se	Permet de préciser les JRE qui peuvent être utilisés par l'application. Les valeurs utilisables par l'attribut version sont 1.2, 1.3 et 1.4. Il est possible de préciser un numéro de version particulier ou d'utiliser le caractère * pour préciser n'importe quel numéro de release. L'ordre des différentes valeurs fournies est important.
Jar	Permet de préciser un fichier .jar qui est utilisé par l'application
Nativelib	Permet de préciser une bibliothèque utilisée par l'application qui contient du code natif
Property	Permet de préciser une propriété système qui sera utilisable par l'application. L'attribut name permet de préciser le nom de la propriété et l'attribut value permet de préciser sa valeur

Le tag <application-desc> permet de préciser la classe qui contient la méthode main() grâce à son attribut main-class.

Nom du tag	Rôle
Argument	Permet de préciser des arguments à l'application tels qu'ils pourraient être fournis sur une ligne de commande

Exemple :

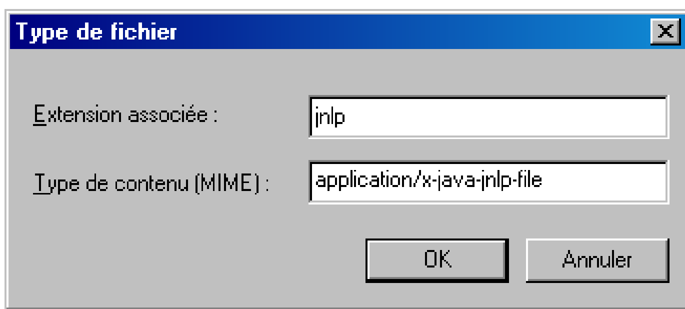
```
<?xml version="1.0" encoding="utf-8"?>
<jnlp spec="1.0+" codebase="http://localhost/" href="MonApplication.jnlp">
  <information>
```

```
    <title>Mon Application</title>
    <vendor>Jean Michel</vendor>
    <homepage href="http://localhost/"/>
    <description>Mon application</description>
    <description kind="short">une application de test</description>
    <offline-allowed/>
  </information>
  <security>
  </security>
  <resources>
    <j2se version="1.4"/>
    <jar href="MonApplication.jar"/>
  </resources>
  <application-desc main-class="com.jmdoudoux.dej.jnlp.MonApplication" />
</jnlp>
```

63.4. La configuration du serveur web

Le serveur qui va servir les fichiers doit être configuré pour qu'il associe le type MIME « application/x-java-jnlp-file » avec l'extension .jnpl

Par exemple sous IIS 5, il faut utiliser l'option propriété du menu contextuel du site. Dans l'onglet « En-Tête http », cliquez sur le bouton « Types de fichiers ». Dans la boîte de dialogue « Type de fichiers », cliquez sur le bouton « Nouveau type » si l'association n'est pas présente dans la liste. Une boîte de dialogue permet de saisir l'extension et le type MIME



Le type MIME permet au navigateur de connaître l'application qui devra être utilisée lors de la réception des données du serveur web.

63.5. Le fichier HTML

Hormis le code mininum requis par la norme HTML, la seule chose indispensable est un lien dont l'URL pointe vers le fichier .jnpl sur le serveur web.

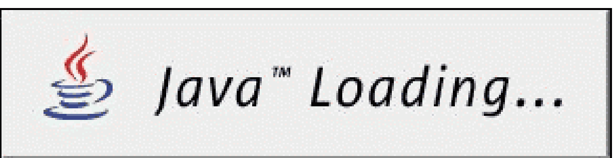
Exemple :

```
<html>
<head>
<title>Mon Application</title>
</head>
<body>
<H1>Mon Application</H1>
<a href="http://localhost/Monapplication.jnlp">Lancez MonApplication</a>
</body>
</html>
```

63.6. Le test de l'application

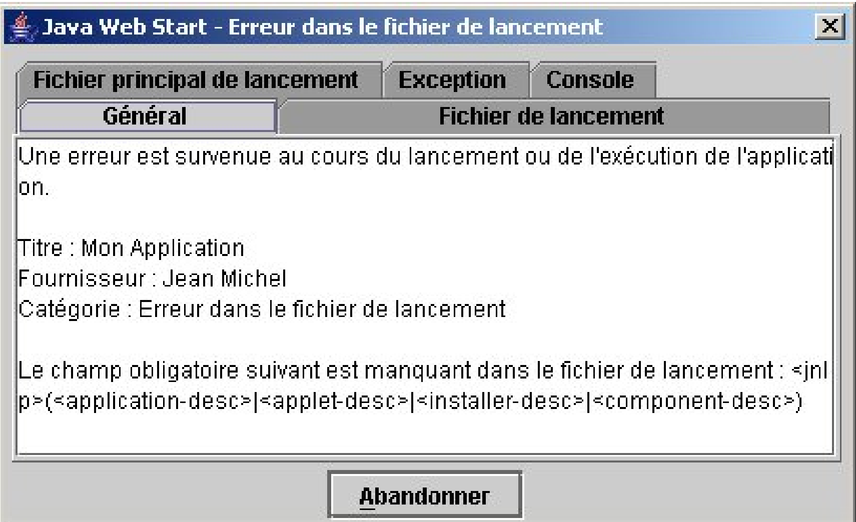
Il faut ouvrir un navigateur et saisir l'url de la page contenant le lien vers le fichier jnlp

Java Web Start se lance



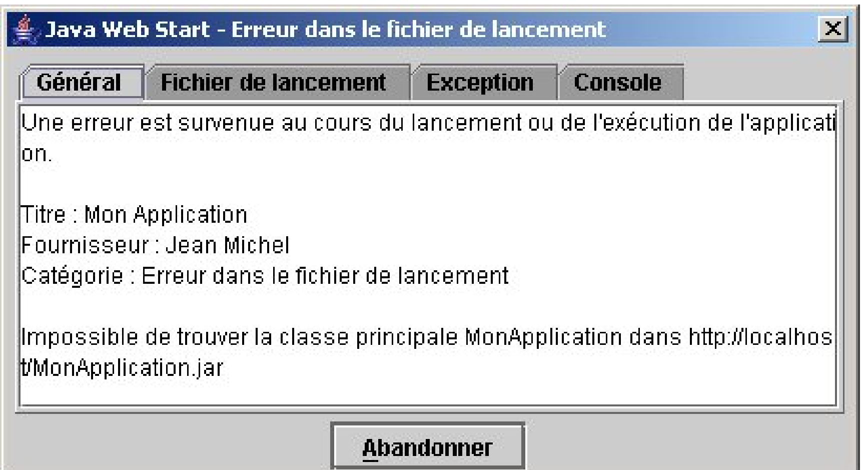
Si le fichier jnlp contient une erreur alors un message d'erreur est affiché.

Cliquez sur « Détails » pour obtenir des informations sur l'erreur.



Si l'application nécessite un accès au système et que le fichier jar n'est pas signé, alors un message erreur est affiché :

Si la classe précisée n'est pas trouvée dans le fichier jar indiqué alors un message d'erreur est affiché



Dans cet exemple, pour résoudre le problème il faut indiquer le nom pleinement qualifié de la classe.

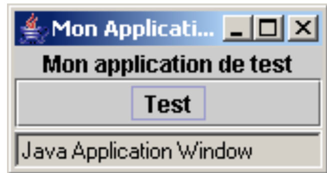
Au premier démarrage réussi d'une application, JWS demande si l'on souhaite créer un raccourci sur le bureau.

En cliquant sur le bouton «Oui», JWS créé un raccourci sur le bureau.

Exemple de raccourci :

```
"C:\Program Files\Java\j2re1.4.2_02\javaws\javaws.exe"
"@C:\Documents and Settings\administrateur\Application Data\
Sun\Java\Deployment\javaws\cache\indirect\indirect31560.ind"
```

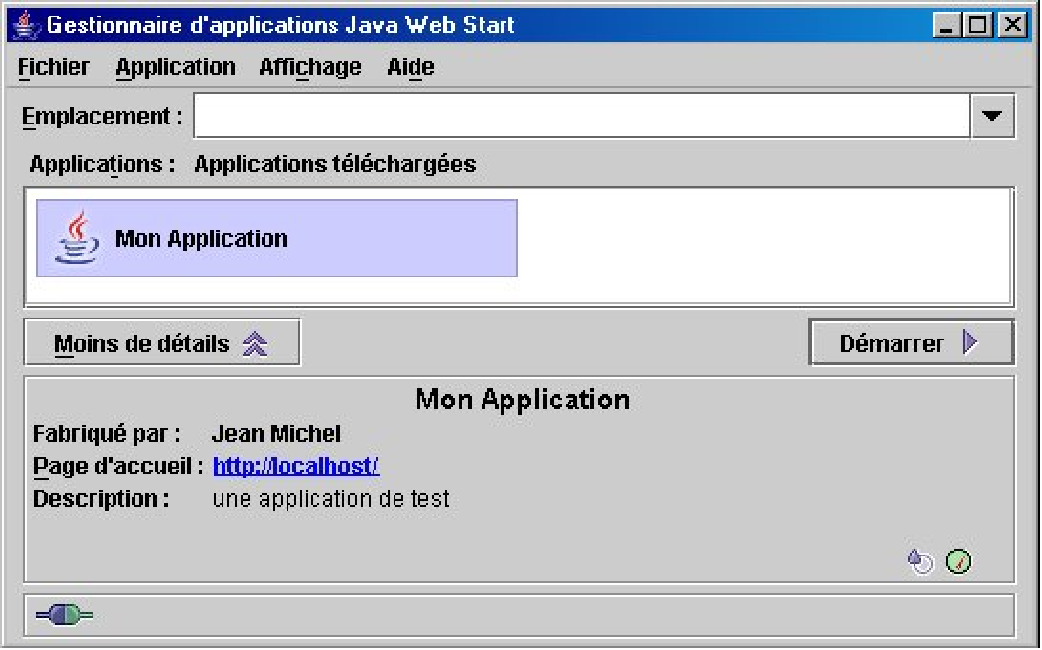
L'application se lance



Comme pour les applets, par mesure de sécurité, un petit libellé en bas des fenêtres est affiché indiquant que la fenêtre est issue de l'exécution d'une application Java.

63.7. L'utilisation du gestionnaire d'applications

Pour lancer le gestionnaire d'applications, il suffit de double cliquer sur l'icône de « Java Web Start » sur le bureau.



Le gestionnaire d'application permet de gérer les applications en local : il permet de lancer les applications déjà téléchargées sur le poste et les mettre à jour.

Plusieurs petites icônes peuvent apparaître selon le contexte

: une mise à jour de l'application est téléchargeable sur le serveur

: l'application peut être exécutée sans connexion au réseau

: l'application est mise en cache en local

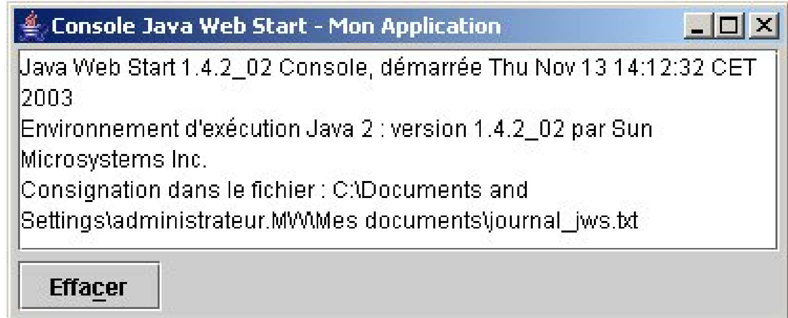
: l'application n'est pas signée

63.7.1. Le lancement d'une application

Pour lancer l'application, il suffit de sélectionner l'application concernée et de cliquer sur le bouton « Démarrer ».

63.7.2. L'affichage de la console

Dans les préférences, sur l'onglet « Avancé », cocher la case à cocher « Afficher la console Java »



63.7.3. Consigner les traces d'exécution dans un fichier de log

Il permet aussi de configurer JWS. Par exemple, en cas de problème, il est possible de demander de consigner une trace d'exécution dans un fichier journal. Celui est particulièrement utile lors du débogage.

Il est possible d'enregistrer les actions dans un fichier de log. Pour cela, il faut cocher la case « Consigner les sorties » et cliquer sur le bouton « Choisir le nom du fichier journal » pour sélectionner ou saisir le nom du fichier.

Exemple :

```
Java Web Start 1.4.2_02 Console, démarrée Thu Nov 13 13:54:36 CET 2003
Environnement d'exécution Java 2 : version 1.4.2_02 par Sun Microsystems Inc.
Consignation dans le fichier : C:\Documents and Settings\admin\Mes documents\journal_jws.txt
Java Web Start 1.4.2_02 Console, démarrée Thu Nov 13 13:54:41 CET 2003
Environnement d'exécution Java 2 : version 1.4.2_02 par Sun Microsystems Inc.
Consignation dans le fichier : C:\Documents and Settings\admin\Mes documents\journal_jws.txt
Java Web Start 1.4.2_02 Console, démarrée Thu Nov 13 13:55:14 CET 2003
Environnement d'exécution Java 2 : version 1.4.2_02 par Sun Microsystems Inc.
Consignation dans le fichier : C:\Documents and Settings\admin\Mes documents\journal_jws.txt
java.lang.NullPointerException
            at sun.misc.Launcher$AppClassLoader.loadClass(Unknown Source)
            at java.lang.ClassLoader.loadClass(Unknown Source)
            at java.lang.ClassLoader.loadClass(Unknown Source)
            at com.sun.javaws.Launcher.continueLaunch(Unknown Source)
            at com.sun.javaws.Launcher.handleApplicationDesc(Unknown Source)
            at com.sun.javaws.Launcher.handleLaunchFile(Unknown Source)
            at com.sun.javaws.Launcher.run(Unknown Source)
            at java.lang.Thread.run(Unknown Source)
```

63.8. L'API de Java Web Start

La suite de ce chapitre sera développée dans une version future de ce document

64. Ajax

Le terme AJAX est l'acronyme de "Asynchronous JavaScript and XML", utilisé pour la première fois par Jesse James Garrett dans son article " Ajax: A New Approach to Web Applications". Ce terme c'est depuis popularisé.

Cependant AJAX est un acronyme qui reflète relativement mal ce qu'est AJAX en réalité : Ajax est un concept qui n'est pas lié particulièrement à un langage de développement et à un format d'échange de données. Cependant, pour faciliter la portabilité, la mise en oeuvre courante d'AJAX fait appel aux technologies Javascript et XML.

AJAX n'est pas une technologie mais plutôt une architecture technique qui permet d'interroger un serveur de manière asynchrone pour obtenir des informations permettant de mettre à jour dynamiquement la page HTML en manipulant son arbre DOM via DHTML.

Ajax est donc un modèle technique dont la mise en oeuvre intègre généralement plusieurs technologies :

- Une page web faisant usage d'Ajax (HTML/CSS/Javascript)
- Une communication asynchrone avec un serveur pour obtenir des données
- Une manipulation de l'arbre DOM de la page pour permettre sa mise à jour (DHTML)
- Une utilisation d'un langage de script (Javascript généralement) pour réaliser les différentes actions côté client

Historiquement, les développeurs d'applications web concentrent leurs efforts sur la partie métier et la persistance des données. La partie IHM est souvent délaissée essentiellement en invoquant les limitations de la technologie HTML. La maturité des technologies misent en oeuvre par le DHTML permettent maintenant de développer des applications plus riches et plus dynamiques.

Le but principal d'Ajax est d'éviter le rechargement complet d'une page pour n'en mettre qu'une partie à jour. Ceci permet donc d'améliorer l'interactivité et le dynamisme de l'application web qui le met en oeuvre.

Le fait de pouvoir opérer des actions asynchrones côté serveur et des mises à jour partielles d'une page web permet d'offrir de nombreuses possibilités de fonctionnalités :

- Rafraîchissement de données : par exemple rafraîchir le contenu d'une liste lors d'une pagination
- Auto-complétion d'une zone de saisie
- Validation de données en temps réel
- Modifier les données dans une table sans utiliser une page dédiée pour faire la mise à jour. Lors du clic sur un bouton modifier, il est possible de transformer les zones d'affichage en zones de saisie, d'utiliser Ajax pour envoyer une requête de mise à jour côté serveur à la validation par l'utilisateur et de réafficher les données modifiées à la place des zones de saisies
- ...

Les utilisations d'Ajax sont donc nombreuses mais cette liste n'est pas exhaustive : cependant elle permet déjà de comprendre qu'AJAX peut rendre les applications web plus dynamiques et interactives.

Les technologies requises pour mettre en oeuvre Ajax sont disponibles depuis plusieurs années (les concepts proposés par Ajax ne sont pas récents puisque Microsoft proposait déjà une solution équivalente dans Internet Explorer 5).

La mise à disposition de ces concepts dans la plupart des navigateurs récents permet à Ajax de connaître un énorme engouement essentiellement justifié par les fonctionnalités proposées et par des mises en oeuvre concrètes à succès des sites tels que Google Gmail, Google Suggest ou Google GMaps. Cet engouement va jusqu'à qualifier de façon générale l'utilisation d'Ajax et quelques autres concepts sous le terme Web 2.0.

Le succès d'AJAX est assuré par le fait qu'il apporte aux utilisateurs d'applications web des fonctionnalités manquantes dans ce type d'applications mais déjà bien connues des utilisateurs dans des applications de type standalone. La mise à jour dynamique de la page apporte aux utilisateurs une convivialité et une rapidité dans les applications web.

L'accroissement de l'utilisation d'AJAX permet de voir apparaître des frameworks qui facilitent sa mise en oeuvre et son intégration dans les applications. Un de ces framework, le framework DWR, est présenté dans ce chapitre.

Le Java Blueprintsde Sun recense les meilleures pratiques d'utilisation d'Ajax avec J2EE : chaque référence propose une description, une solution de conception et un exemple de code fonctionnel mettant en oeuvre la solution. Ces références sont actuellement l'auto-complétion, une barre de progression et la validation des données d'un formulaire.

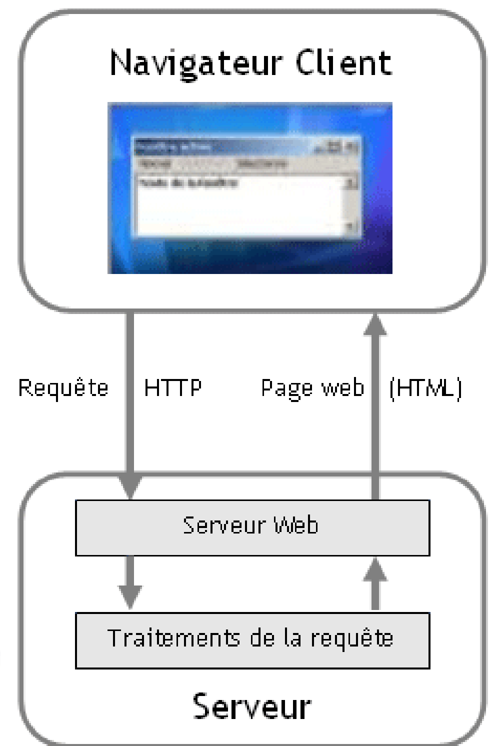
Ajax est aussi en cours d'intégration dans les Java Server Faces.

64.1. La présentation d'Ajax

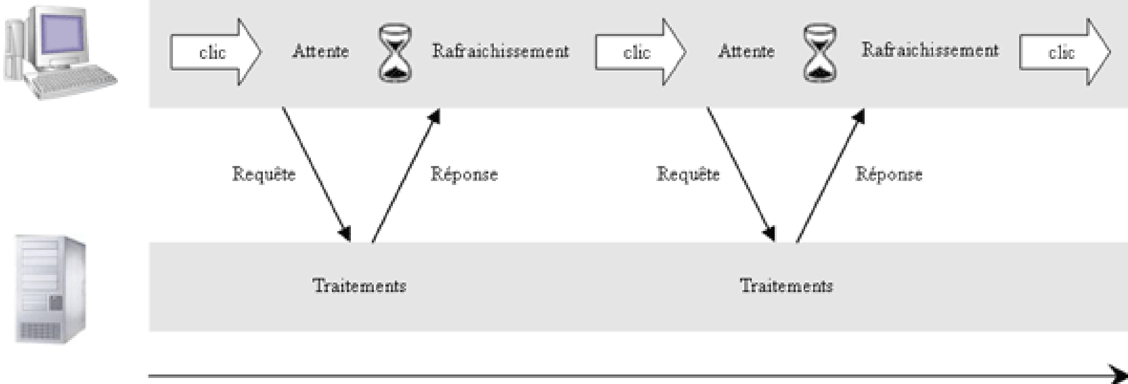
Traditionnellement les pages HTML ont besoin d'être entièrement rafraîchies dès lors qu'une simple portion de la page doit être rafraîchie. Ce mode de fonctionnement possède plusieurs inconvénients :

- limite les temps de réponse de l'application,
- augmente la consommation de bande passante et de ressources côté serveur
- perte du contexte lié au protocole http (utilisation de mécanisme tel que les cookies ou la session pour conserver un état)

Dans une application web, les échanges entre le client et le serveur sont opérés de manière synchrone. Chaque appel nécessitant un traitement côté serveur impose un rafraîchissement complet de la page. Le mode de fonctionnement d'une application web classique est donc le suivant :



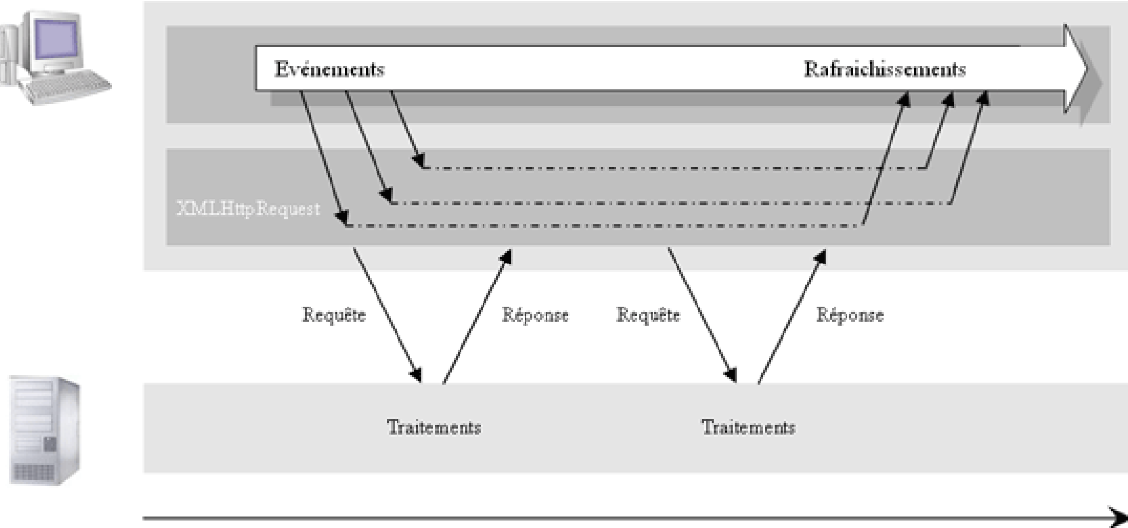
Avant Ajax, les applications web fonctionnaient sur un mode soumission/attente/rafraîchissement totale de la page. Chaque appel au serveur retourne la totalité de la page qui est donc entièrement reconstruite et retourné au navigateur pour affichage. Durant cette échange, l'utilisateur est obligé d'attendre la réponse du serveur ce qui implique au mieux un clignotement lors du rafraîchissement de la page ou l'affichage d'une page blanche en fonction du temps de traitement de la requête par le serveur.



Ajax repose essentiellement sur un échange asynchrone entre le client et le serveur ce qui évite aux utilisateurs d'avoir un temps d'attente obligatoire entre leur action et la réponse correspondante tant qu'il reste dans la même page. Ce mode de communication et le rafraîchissement partiel de la page en fonction des données reçues en réponse du serveur permettent d'avoir une meilleure réactivité aux actions de l'utilisateur.

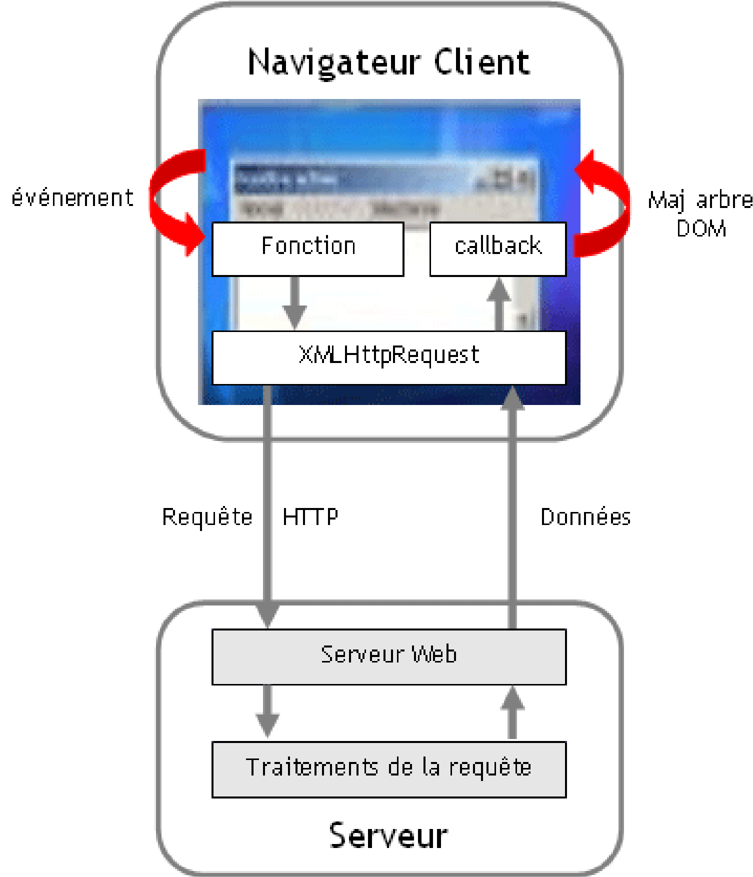
Avec Ajax :

- Le rafraîchissement partiel d'une page remplace le rafraîchissement systématique total de la page
- La communication asynchrone remplace la communication synchrone entre le client et le serveur. Ceci permet d'améliorer l'interactivité entre l'utilisateur et l'application



L'utilisation d'Ajax dans une application web permet des communications asynchrones, ce qui permet à l'utilisateur de rester dans la page courante. La mise à jour dynamique de la page en fonction de la réponse permet de rendre ces traitements transparents pour l'utilisateur et surtout de lui donner une impression de fluidité.

Le mode de fonctionnement d'une application web utilisant Ajax est le suivant :



Tous ces traitements sont déclenchés par un événement utilisateur sur un composant (clic, changement d'une valeur, perte du focus, ...) ou système (timer, ...) dans la page.

La partie centrale d'Ajax est un moteur capable de communiquer de façon asynchrone avec un serveur en utilisant le protocole http. Généralement les appels au serveur se font via l'objet Javascript XMLHttpRequest. Cet objet n'est pas défini dans les spécifications courantes de Javascript mais il est implémenté dans tous les navigateurs récents car il devient un standard de facto. Il est donc important de noter qu'Ajax ne fonctionnera pas sur des navigateurs anciens : ceci est à prendre en compte lors d'une volonté d'utilisation d'Ajax ou lors de sa mise en oeuvre.

L'objet Javascript XMLHttpRequest occupe donc un rôle majeur dans Ajax puisqu'il assure les communications entre le client et le serveur. Généralement ces communications sont asynchrones pour permettre à l'utilisateur de poursuivre ces activités dans la page.

AJAX nécessite une architecture différente côté serveur : habituellement en réponse à une requête le serveur renvoie le contenu de toute la page. En réponse à une requête faite via AJAX, le serveur doit renvoyer des informations qui seront utilisées côté client par du code Javascript pour mettre à jour la page. Le format de ces informations est généralement XML mais ce n'est pas une obligation.

Le rafraîchissement partiel d'une page via Ajax permet d'accroître la réactivité de l'IHM mais aussi de diminuer la bande passante et les ressources serveurs consommées lors d'un rafraîchissement complet de la page web.

La mise à jour partielle d'une page en modifiant directement son arbre DOM permet de conserver le contexte de l'état de la page. Les parties inchangées via le DOM restent inchangées et sont toujours connues et utilisables. Cependant un des effets pervers pour l'utilisateur est l'utilisation du bouton back du navigateur dont l'utilisateur est habitué à obtenir l'état précédent de la page dans le cas d'un rafraîchissement à chaque action. Ce nouveau mode peut être déroutant pour l'utilisateur.

Pour se donner une idée simple des puissantes possibilités offertes par Ajax, un exemple peut être offert en utilisant Google Suggest à l'url :

http://www.google.com/webhp?complete=1&hl=fr

java

javascript	520,000,000 résultats
java.com	2,440,000 résultats
java runtime	2,680,000 résultats
java sun	11,400,000 résultats
javascript telecharger	3,060,000 résultats
java download	14,400,000 résultats
javaboy	38,500 résultats
java update	6,260,000 résultats
javadoc	6,690,000 résultats
java runtime environment	1,850,000 résultats

fermer

Au fur et à mesure de la saisie de caractères dans la zone de texte, une liste déroulante propose des suggestions avec le nombre de résultats obtenus correspondant dans le moteur de recherche Google.

64.2. Le détail du mode de fonctionnement

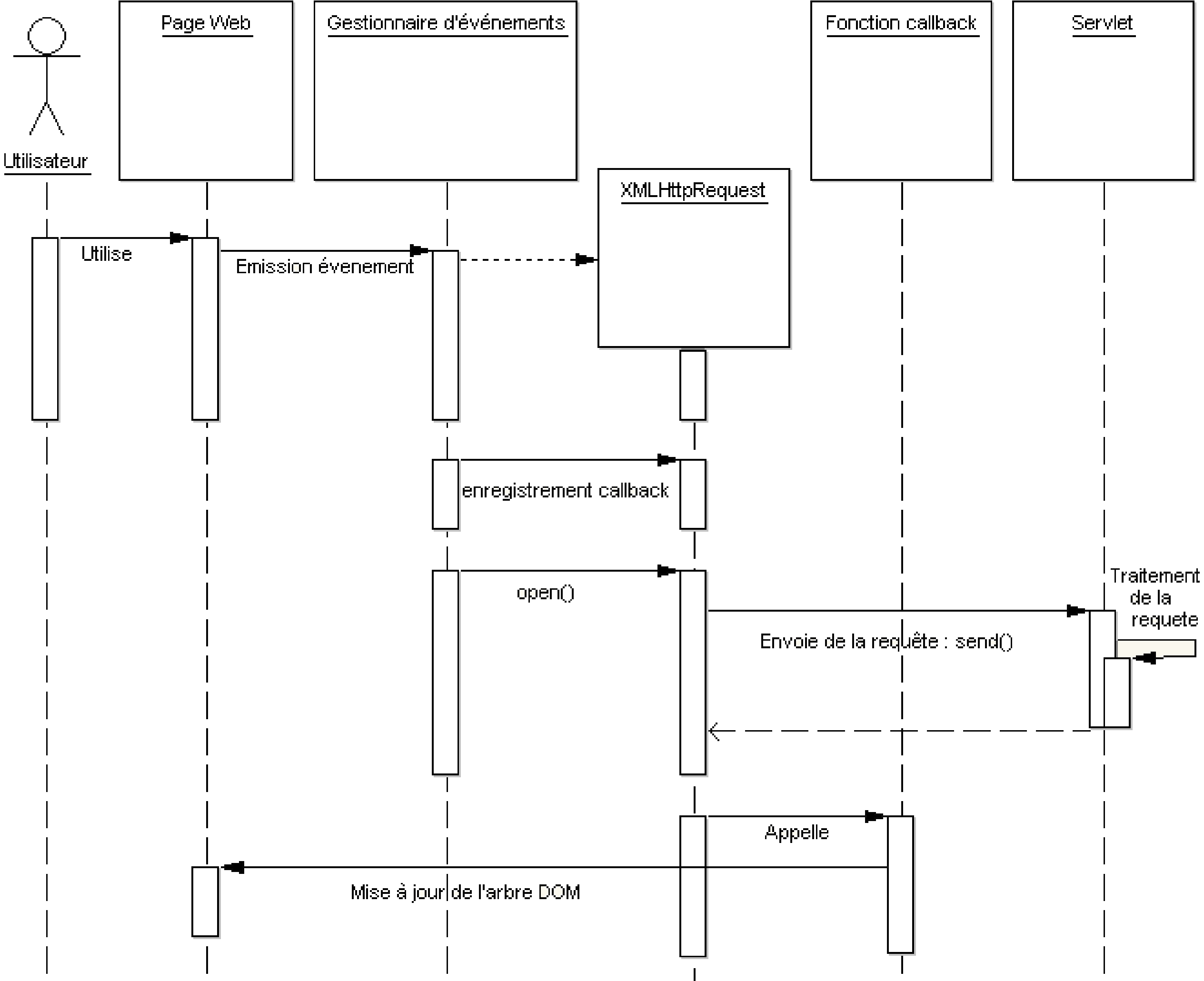
La condition pour utiliser Ajax dans une application web est que le support de Javascript soit activé dans le navigateur et que celui-ci propose une implémentation de l'objet XMLHttpRequest.

L'objet XMLHttpRequest permet un échange synchrone ou asynchrone avec le serveur en utilisant le protocole HTTP. La requête http envoyée au serveur peut être de type GET ou POST.

Une communication asynchrone permet au navigateur de ne pas bloquer les actions de l'utilisateur en attendant la réponse du serveur. Ainsi, une fonction de type callback est enregistrée pour permettre son appel à la réception de la réponse http.

Côté serveur toute technologie permettant de répondre à une requête http peut être utilisée avec Ajax. J2EE et plus particulièrement les servlets se prêtent particulièrement bien à ces traitements. La requête http est traitée comme toutes les <frequêtes de ce type. En fonction des paramètres reçus de la requête, des traitements sont exécutés pour générer la réponse http.

A la réception de la réponse par le client, la fonction de type callback est appelée. Elle se charge d'extraire les données de la réponse et réaliser les traitements de mise à jour de la page web en manipulant son arbre DOM.



Les avantages d'AJAX sont :

- Une économie de ressources côté serveur et de bande passante puisque la page n'est pas systématiquement transmise pour une mise à jour
- Une meilleure réactivité et dynamique de l'application web

AJAX possède cependant quelques inconvénients :

- Complexité liée à l'utilisation de plusieurs technologies côté client et serveur
- Utilisation de Javascript : elle implique la prise en compte des inconvénients de cette technologie : difficulté pour déboguer, différences d'implémentation selon le navigateur, code source visible, ...
- AJAX ne peut être utilisé qu'avec des navigateurs possédant une implémentation de l'objet XMLHttpRequest
- L'objet XMLHttpRequest n'est pas standardisé ce qui nécessite des traitements Javascript dépendant du navigateur utilisé
- Le changement du mode de fonctionnement des applications web (par exemple : impossible de faire un favori vers une page dans un certain état, le bouton back ne permet plus de réafficher la page dans son état précédent la dernière action, ...)
- La mise en oeuvre de nombreuses fonctionnalités mettant en oeuvre Ajax peut faire rapidement augmenter le nombre de requêtes http à traiter par le serveur
- Le manque de frameworks et d'outils pour faciliter la mise en oeuvre

Ajax possède donc quelques inconvénients qui nécessitent une sérieuse réflexion pour une utilisation intensive dans une application. Un bon compromis est d'utiliser Ajax pour des fonctionnalités permettant une amélioration de l'interactivité entre l'application et l'utilisateur.

Actuellement, Ajax et en particulier l'objet XMLHttpRequest n'est pas un standard. De plus, reposant essentiellement sur Javascript, son bon fonctionnement ne peut pas être assuré sur tous les navigateurs. Pour ceux avec qui cela peut l'être, le support de Javascript doit être activé et il est quasiment impératif d'écrire du code dépendant du navigateur utilisé.

Il peut donc être nécessaire de prévoir, lors du développement de l'application, le bon fonctionnement de cette dernière sans utiliser Ajax pour permettre notamment un fonctionnement correct sur les anciens navigateurs ou sur les navigateurs où le support de Javascript est désactivé.

Le plus simple pour assurer cette tâche est de détecter au démarrage de l'application si l'objet XMLHttpRequest est utilisable dans le navigateur de l'utilisateur. Dans l'affirmative, l'application renvoie une version avec Ajax de la page sinon une version sans Ajax.

Comme la requête est asynchrone, il est peut être important d'informer l'utilisation sur l'état des traitements en cours et surtout sur le succès ou l'échec de leur exécution. Avec un rafraîchissement traditionnel complet de la page c'est facile. En utilisant Ajax, il est nécessaire de faire usage de subtilités d'affichage ou d'effets visuels auxquels l'utilisateur n'est pas forcement habitué. Un exemple concret concerne un bouton de validation : il peut être utile de modifier le libellé du bouton pour informer l'utilisateur que les traitements sont en cours afin d'éviter qu'il clic plusieurs fois sur le bouton.

Il faut aussi garder à l'esprit que les échanges asynchrones ne garantissent pas que les réponses arrivent dans le même ordre que les requêtes correspondantes sont envoyées. Il est même tout à fait possible de ne jamais recevoir une réponse. Il faut donc être prudent pour vouloir enchaîner plusieurs requêtes.

64.3. Un exemple simple

Cet exemple va permettre de réaliser une validation côté serveur d'une donnée saisie en temps réel.

Une servlet permettra de réaliser cette validation. La validation proposée est volontairement simpliste et pourrait même être réalisée directement côté client avec du code Javascript. Il faut cependant comprendre que les traitements de validation pourraient être beaucoup plus complexes avec par exemple une recherche dans une base de données, ce qui justifierait pleinement l'emploi d'une validation côté serveur.

Les actions suivantes sont exécutées dans cet exemple :

- Un événement déclencheur est émis (la saisie d'une donnée par l'utilisateur)
- Création et paramétrage d'un objet de type XMLHttpRequest
- Appel de la servlet par l'objet XMLHttpRequest
- La servlet exécute les traitements de validation et renvoie le résultat en réponse au format XML
- L'objet XMLHttpRequest appelle la fonction d'exploitation de la réponse
- La fonction met à jour l'arbre DOM de la page en fonction des données de la réponse

64.3.1. L'application de tests

La page de test est une JSP qui contient un champ de saisie.

Exemple :

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
        pageEncoding="ISO-8859-1"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Test validation AJAX</title>
<script type="text/javascript">
<!--
var requete;

function valider() {
   var donnees = document.getElementById("donnees");
   var url = "valider?valeur=" + escape(donnees.value);
   if (window.XMLHttpRequest) {
       requete = new XMLHttpRequest();
   } else if (window.ActiveXObject) {
       requete = new ActiveXObject("Microsoft.XMLHTTP");
```

```
    }
    requete.open("GET", url, true);
    requete.onreadystatechange = majIHM;
    requete.send(null);
}

function majIHM() {
  var message = "";

  if (requete.readyState == 4) {
    if (requete.status == 200) {
      // exploitation des données de la réponse
      var messageTag = requete.responseXML.getElementsByTagName("message")[0];
      message = messageTag.childNodes[0].nodeValue;
      mdiv = document.getElementById("validationMessage");
      if (message == "invalide") {
         mdiv.innerHTML = "<img src='images/invalide.gif'>";
      } else {
         mdiv.innerHTML = "<img src='images/valide.gif'>";
      }
    }
  }
}

//-->
</script>
</head>
<body>
<table>
        <tr>
                <td>Valeur :</td>
                <td nowrap><input type="text" id="donnees" name="donnees" size="30"
                        onkeyup="valider();"></td>
                <td>
                <div id="validationMessage"></div>
                </td>
        </tr>
</table>
</body>
</html>
```

Le code Javascript est détaillé dans les sections suivantes.

L'application contient aussi une servlet qui sera détaillée dans une des sections suivantes.

Le descripteur de déploiement de l'application contient la déclaration de la servlet.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
     "web-app_2_2.dtd">
<web-app>
        <display-name>Test de validation avec Ajax</display-name>
        <servlet>
                <servlet-name>ValiderServlet</servlet-name>
                <display-name>ValiderServlet</display-name>
                <description>Validation de données</description>
                <servlet-class>
                com.jmd.test.ajax.ValiderServlet</servlet-class>
        </servlet>
        <servlet-mapping>
                <servlet-name>ValiderServlet</servlet-name>
                <url-pattern>/valider</url-pattern>
        </servlet-mapping>
        <welcome-file-list>
                <welcome-file>index.jsp</welcome-file>
        </welcome-file-list>
</web-app>
```

64.3.2. La prise en compte de l'événement déclencheur

Un événement onkeyup est associé à la zone de saisie des données. Cet événement va appeler la fonction Javascript valider().

Exemple :

```
<input type="text" id="donnees" name="donnees" size="30"
              onkeyup="valider();">
```

Ainsi la fonction sera appelée à chaque fois que l'utilisateur saisit un caractère.

64.3.3. La création d'un objet de type XMLHttpRequest pour appeler la servlet

La fonction Javascript valider() va réaliser les traitements de la validation des données.

Exemple :

```
var requete;

function valider() {
   var donnees = document.getElementById("donnees");
   var url = "valider?valeur=" + escape(donnees.value);
   if (window.XMLHttpRequest) {
      requete = new XMLHttpRequest();
      requete.open("GET", url, true);
      requete.onreadystatechange = majIHM;
      requete.send(null);
   } else if (window.ActiveXObject) {
      requete = new ActiveXObject("Microsoft.XMLHTTP");
      if (requete) {
         requete.open("GET", url, true);
         requete.onreadystatechange = majIHM;
         requete.send();
      }
   } else {
      alert("Le navigateur ne supporte pas la technologie Ajax");
   }
}
```

Elle réalise les traitements suivants :

- récupère les données saisies
- détermine l'url d'appel de la servlet en passant en paramètre les données. Ces données sont encodées selon la norme http grâce à la fonction escape().
- instancie une requête de type XMLHttpRequest en fonction du navigateur utilisé
- associe à la requête l'url et la fonction à exécuter à la réponse
- exécute la requête

Comme dans de nombreux usages courants de Javascript, des traitements dépendants du navigateur cible à l'exécution sont nécessaires. Dans le cas de l'instanciation de l'objet XMLHttpRequest, celui-ci est un ActiveX sous Internet Explorer et un objet natif sur les autres navigateurs qui le supportent.

La signature de la méthode open de l'objet XMLHttpRequest est XMLHttpRequest.open(String method, String URL, boolean asynchronous).

Le premier paramètre est le type de requête http réalisé par la requête (GET ou POST)

Le second paramètre est l'url utilisée par la requête.

Le troisième paramètre est un booléen qui précise si la requête doit être effectuée de façon asynchrone. Si la valeur passée est true alors une fonction de type callback doit être associée à l'événement onreadystatechange de la requête. La fonction précisée sera alors exécutée à la réception de la réponse.

La méthode send() permet d'exécuter la requête http en fonction des paramètres de l'objet XMLHttpRequest.

Pour une requête de type GET, il suffit de passer null comme paramètre de la méthode send().

Pour une requête de type POST, il faut préciser le Content-Type dans l'en-tête de la requête et fournir les paramètres en paramètre de la fonction send().

Exemple :

```
requete.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
requete.send("valeur=" + escape(donnees.value));
```

L'objet XMLHttpRequest possède les méthodes suivantes :

Méthode	Rôle
abort()	Abandon de la requête
getAllResponseHeaders()	Renvoie une chaîne contenant les en-têtes http de la réponse
getResponseHeader(nom)	Renvoie la valeur de l'en-tête dont le nom est fourni en paramètre
setTimeouts(duree)	Précise la durée maximale pour l'obtention de la réponse
setRequestHeader(nom, valeur)	Précise la valeur de l'en-tête dont le nom est fournie en paramètre
open(méthode, url, [assynchrone[, utilisateur[, motdepasse]]]	Prépare une requête en précisant la méthode (Get ou Post), l'url, un booléen optionnel qui précise si l'appel doit être asynchrone et le user et/ou le mot de passe optionnel
send(data)	Envoi de la requête au serveur

L'objet XMLHttpRequest possède les propriétés suivantes :

Propriété	Rôle
onreadystatchange	Précise la fonction de type callback qui est appelée lorsque la valeur de la propriété readyState change
readyState	L'état de la requête : 0 = uninitialized 1 = loading 2 = loaded 3 = interactive 4 = complete
responseText	Le contenu de la réponse au format texte
responseXML	Le contenu de la réponse au format XML
status	Le code retour http de la réponse
statusText	La description du code retour http de la réponse

Il peut être intéressant d'utiliser une fonction Javascript qui va générer une chaîne de caractères contenant le nom et la valeur de chacun des éléments d'un formulaire.

Exemple :

```
function getFormAsString(nomFormulaire){

  resultat ="";
  formElements=document.forms[nomFormulaire].elements;

  for(var i=0; i<formElements.length; i++ ){
    if (i > 0) {
      resultat+="&";
    }
    resultat+=escape(formElements[i].name)+"="
    +escape(formElements[i].value);
 }

 return resultat;
}
```

Ceci facilite la génération d'une url qui aurait besoin de toutes les valeurs d'un formulaire.

64.3.4. L'exécution des traitements et le renvoie de la réponse par la servlet

La servlet associée à l'URI valider est exécutée par le conteneur web en réponse à la requête.

Exemple :

```
package com.jmd.test.ajax;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * Servlet ValiderServlet
 *
 */
 public class ValiderServlet extends javax.servlet.http.HttpServlet
                          implements javax.servlet.Servlet {

  /* (non-Java-doc)
         * @see javax.servlet.http.HttpServlet#HttpServlet()
         */
        public ValiderServlet() {
                super();
        }

        /* (non-Java-doc)
         * @see javax.servlet.http.HttpServlet#doGet(HttpServletRequest request,
         *                                            HttpServletResponse response)
         */
        protected void doGet(HttpServletRequest request, HttpServletResponse response)
                             throws ServletException, IOException {
    String resultat = "invalide";
    String valeur = request.getParameter("valeur");

    response.setContentType("text/xml");
    response.setHeader("Cache-Control", "no-cache");

    if ((valeur != null) && valeur.startsWith("X")) {
      resultat = "valide";
    }

    response.getWriter().write("<message>"+resultat+"</message>");
  }
}
```

La validation est assurée si la valeur fournie commence par un caractère "X".

La servlet renvoie simplement un texte indiquant l'état de la validation réalisée dans une balise message.

Il est important que le type Mime retournée dans la réponse soit de type "text/xml".

Il est préférable de supprimer la mise en cache de la réponse par le navigateur. Cette suppression est obligatoire si une même requête peut renvoyer une réponse différente lors de plusieurs appels.

64.3.5. L'exploitation de la réponse

L'objet XMLHttpRequest appelle la fonction de type callback majIHM() à chaque fois que la propriété readyState change de valeur.

La fonction majIHM() commence donc par vérifier la valeur de la propriété readyState. Si celle-ci vaut 4 alors l'exécution de la requête est complète.

Dans ce cas, il faut vérifier le code retour de la réponse http. La valeur 200 indique que la requête a correctement été traitée.

Exemple :

```
function majIHM() {
  var message = "";

  if (requete.readyState == 4) {
    if (requete.status == 200) {
      // exploitation des données de la réponse
      // ...
    } else {
       alert('Une erreur est survenue lors de la mise à jour de la page');
    }
  }
}
```

La fonction modifie alors le contenu de la page en modifiant son arbre DOM en utilisant la valeur de la réponse. Cette valeur au format XML est obtenue en utilisant la fonction responseXML de l'instance de XMLHttpRequest. La valeur au format texte brut peut être obtenue en utilisant la fonction responseText.

Il est alors possible d'exploiter les données de la réponse.

Exemple :

```
function majIHM() {
  var message = "";

  if (requete.readyState == 4) {
    if (requete.status == 200) {
      // exploitation des données de la réponse
      var messageTag = requete.responseXML.getElementsByTagName("message")[0];
      message = messageTag.childNodes[0].nodeValue;
      mdiv = document.getElementById("validationMessage");
      if (message == "invalide") {
         mdiv.innerHTML = "<img src='images/invalide.gif'>";
      } else {
         mdiv.innerHTML = "<img src='images/valide.gif'>";
      }
    } else {
      alert('Une erreur est survenue lors de la mise à jour de la page.'+
                '\n\nCode retour = '+requete.statusText);
    }
  }
}
```

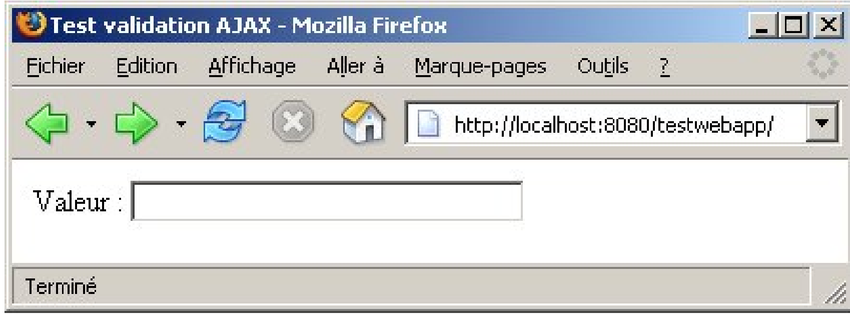
Il est aussi possible que la réponse contienne directement du code HTML à afficher. Il suffit simplement d'affecter le résultat de la réponse au format texte à la propriété innerHTML de l'élément de la page à rafraîchir.

Exemple :

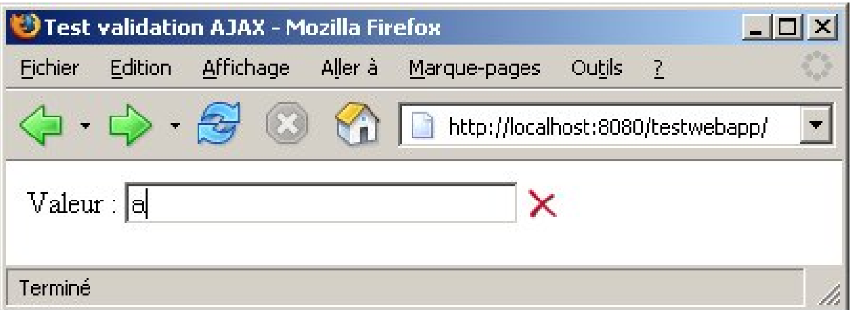
```
function majIHM() {
  if (requete.readyState == 4) {
    if (requete.status == 200) {
      document.getElementById("validationMessage").innerHTML = requete.responseText;
    } else {
      alert('Une erreur est survenue lors de la mise à jour de la page.'+
                 '\n\nCode retour = '+requete.statusText);
  }
}
```

64.3.6. L'exécution de l'application

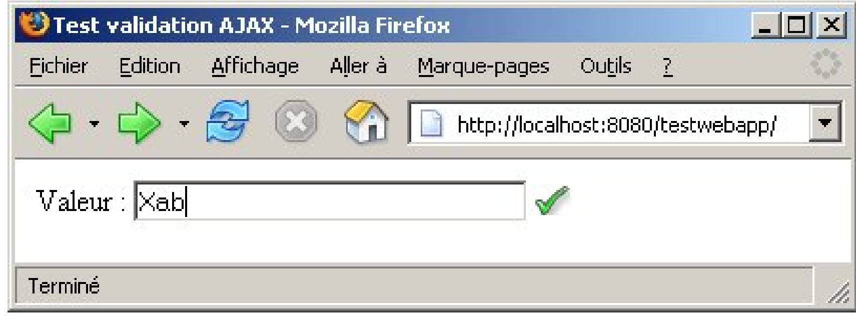
La page de tests s'affiche au lancement de l'application



La saisie d'un caractère déclenche la validation



L'icône dépend du résultat de la validation.



64.4. Des frameworks pour mettre en oeuvre Ajax

La mise en oeuvre directe de l'objet XmlHttpRequest est relativement lourde (nécessite l'écriture de nombreuses lignes de code), fastidieuse (pas facile à déboguer) et souvent répétitive. La mise en oeuvre de plusieurs technologies côté client et serveur peut engendrer de nombreuses difficultés notamment dans le code Javascript (débogage difficile, gestion de la compatibilité du support par les navigateurs, ...).

Aussi de nombreux frameworks commencent à voir le jour pour faciliter le travail des développeurs. Cette section va détailler l'utilisation du framework DWR et proposer une liste non exhaustive d'autres frameworks.

64.4.1. Direct Web Remoting (DWR)

DWR (Direct Web Remoting) est une bibliothèque open source Java dont le but est de faciliter la mise en oeuvre d'Ajax dans les applications Java.

DWR se charge de générer le code Javascript permettant l'appel à des objets Java de type bean qu'il suffit d'écrire. Sa devise est "Easy Ajax for Java".

DWR encapsule les interactions entre le code Javascript côté client et les objets Java côté serveur : ceci rend transparent l'appel de ces objets côté client.

La mise en oeuvre de DWR côté serveur est facile :

- Ajouter le fichier dwr.jar au classpath de l'application
- Configurer une servlet dédiée aux traitements des requêtes dans le fichier web.xml
- Ecrire les beans qui seront utilisés dans les pages
- Définir ces beans dans un fichier de configuration de DWR

La mise en oeuvre côté client nécessite d'inclure des bibliothèques Javascript générées dynamiquement par la servlet de DWR. Il est alors possible d'utiliser les fonctions Javascript générées pour appeler les méthodes des beans configurés côté serveur.

DWR s'intègre facilement dans une application web puisqu'il repose sur une servlet et plus particulièrement avec celles mettant en oeuvre le framework Spring dont elle propose un support. DWR est aussi inclus dans le framework WebWork depuis sa version 2.2.

DWR fournit aussi une bibliothèque Javascript proposant des fonctions de manipulations courantes en DHTML : modifier le contenu des conteneurs <DIV> ou <SPAN>, remplir une liste déroulante avec des valeurs, etc ...

DWR est une solution qui encapsule l'appel de méthodes de simples objets de type Javabean exécutés sur le serveur dans du code Javascript généré dynamiquement. Le grand intérêt est de masquer toute la complexité de l'utilisation de l'objet XMLHttpRequest et de simplifier à l'extrême le code à développer côté serveur.

DWR se compose de deux parties :

- Du code Javascript qui envoie des requêtes à la servlet et met à jour la page à partir des données de la réponse
- Une servlet qui traite les requêtes reçues et renvoie une réponse au navigateur

Côté serveur, une servlet est déployée dans l'application web. Cette servlet a deux rôles principaux :

1. Elle permet de générer dynamiquement des bibliothèques de code Javascript. Deux de celles-ci sont à usage général. Une bibliothèque de code est générée pour chaque bean défini dans la configuration de DWR
2. Elle permet de traiter les requêtes émises par le code Javascript générés pour appeler la méthode d'un bean

DWR génère dynamiquement le code Javascript à partir des Javabeans configurés dans un fichier de paramètres en utilisant l'introspection. Ce code se charge d'encapsuler les appels aux méthodes du bean, ceci incluant la conversion du format des données de Javascript vers Java et vice et versa. Ce mécanisme est donc similaire à d'autres solutions de type RPC (remote procedure call).

Une fonction de type callback est précisée à DWR pour être exécutée à la réception de la réponse de la requête vers la méthode d'un bean.

DWR facilite donc la mise en oeuvre d'Ajax avec Java côté serveur : il se charge de toute l'intégration de Javabeans côté serveur pour permettre leur appel côté client de manière transparente.

Le site officiel de DWR est à l'url : http://getahead.ltd.uk/dwr/ ou https://dwr.dev.java.net/

La documentation de ce projet est particulièrement riche et de nombreux exemples sont fournis sur le site.

La version utilisée dans cette section est la version 1.1.1. Elle nécessite un JDK 1.3 et conteneur web supportant la version 2.2 de l'API servlet.

64.4.1.1. Un exemple de mise en oeuvre de DWR

Il faut télécharger le fichier dwr.jar sur le site officiel de DWR et l'ajouter dans le répertoire WEB-INF/Lib de l'application web qui va utiliser la bibliothèque.

Il faut ensuite déclarer dans le fichier de déploiement de l'application web.xml la servlet qui sera utilisée par DWR. Il faut déclarer la servlet et définir son mapping :

Exemple :

```
        <servlet>
                <servlet-name>dwr-invoker</servlet-name>
                <display-name>DWR Servlet</display-name>
                <servlet-class>uk.ltd.getahead.dwr.DWRServlet</servlet-class>

                <init-param>
                        <param-name>debug</param-name>
                        <param-value>true</param-value>
                </init-param>
        </servlet>

        <servlet-mapping>
                <servlet-name>dwr-invoker</servlet-name>
                <url-pattern>/dwr/*</url-pattern>
        </servlet-mapping>
```

Il faut créer un fichier de configuration pour DWR nommé dwr.xml dans le répertoire WEB-INF de l'application

Exemple :

```
<!DOCTYPE dwr PUBLIC
    "-//GetAhead Limited//DTD Direct Web Remoting 1.0//EN"
    "http://www.getahead.ltd.uk/dwr/dwr10.dtd">
<dwr>
  <allow>
    <create creator="new" javascript="JDate">
      <param name="class" value="java.util.Date"/>
    </create>
  </allow>
</dwr>
```

Ce fichier permet de déclarer à DWR la liste des beans qu'il devra encapsuler pour des appels via Javascript. Dans l'exemple, c'est la classe java.util.Date fourni dans l'API standard qui est utilisée.

Le creator de type "new" instancie la classe en utilisant le constructeur sans argument. L'attribut javascript permet de préciser le nom de l'objet Javascript qui sera utilisé côté client.

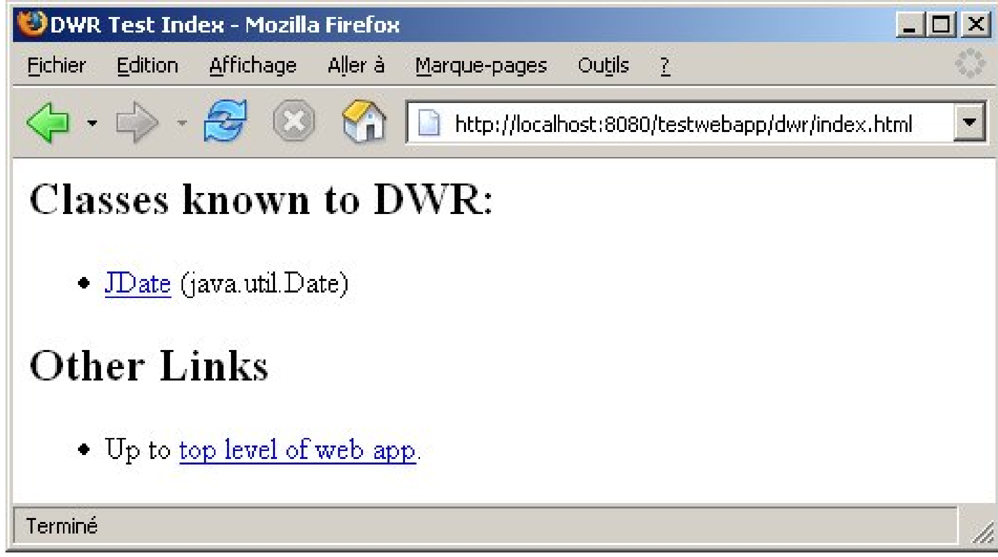
Le tag param avec l'attribut name ayant pour valeur class permet de préciser le nom pleinement qualifié du Bean à encapsuler.

DWR possède quelques restrictions :

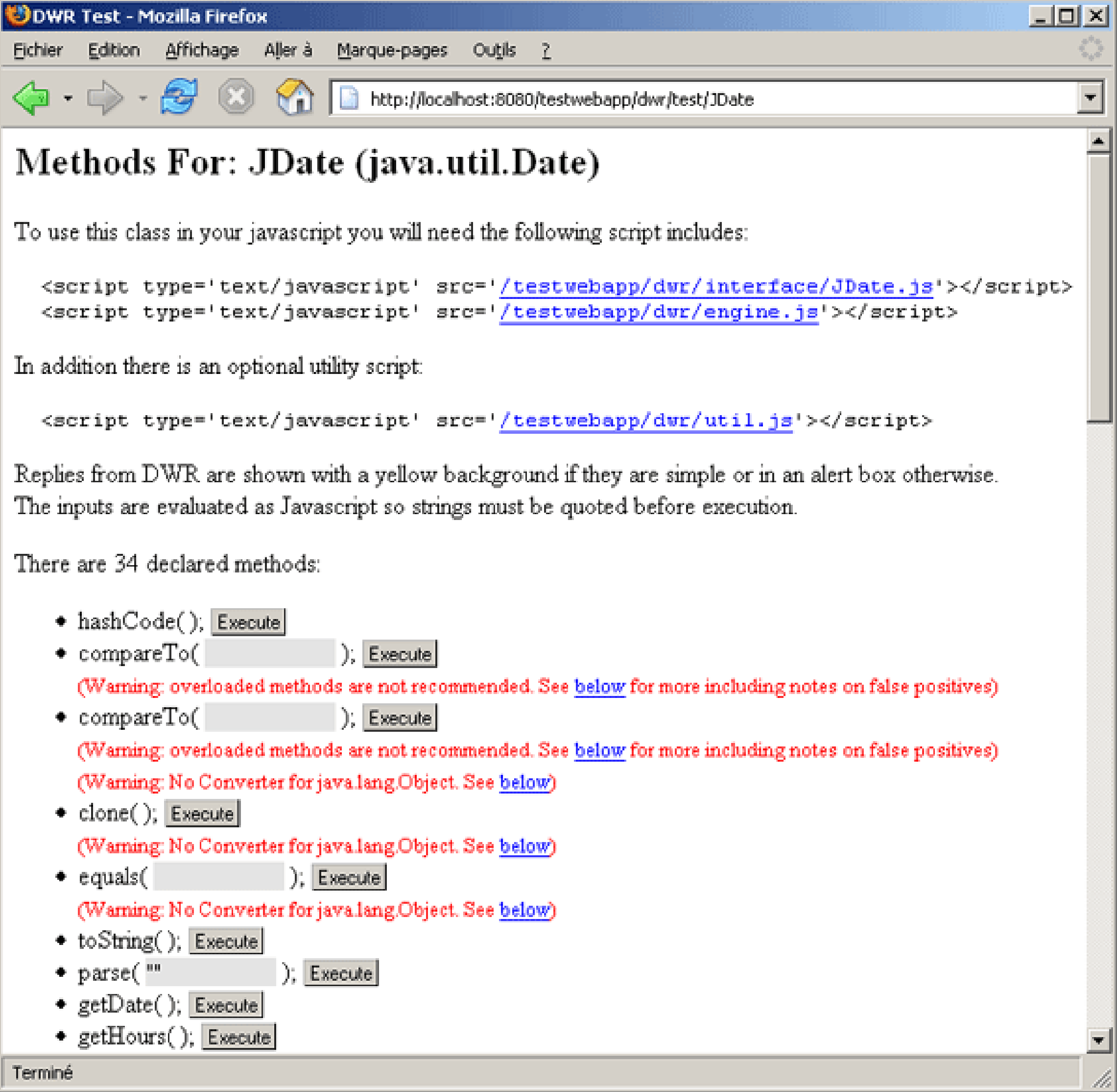
- Il ne faut surtout pas utiliser de nom de méthode dans les beans exposés correspondant à des mots réservés en Javascript. Un exemple courant est le mot delete
- Il faut éviter l'utilisation de méthodes surchargées

Par défaut, DWR encapsule toutes les méthodes public de la classe définie. Il est donc nécessaire de limiter les méthodes utilisables via DWR à celles requises par les besoins de l'application soit dans la définition des membres de la classe soit dans le fichier de configuration de DWR.

Il suffit alors de lancer l'application et d'ouvrir un navigateur sur l'url de l'application et d'ajouter /dwr



Cette page liste tous les beans qui sont encapsulés par DWR. Il suffit de cliquer sur le lien d'un bean pour voir afficher une page de test de ce bean. Cette page génère dynamiquement une liste de toutes les méthodes pouvant être appelées en utilisant DWR.



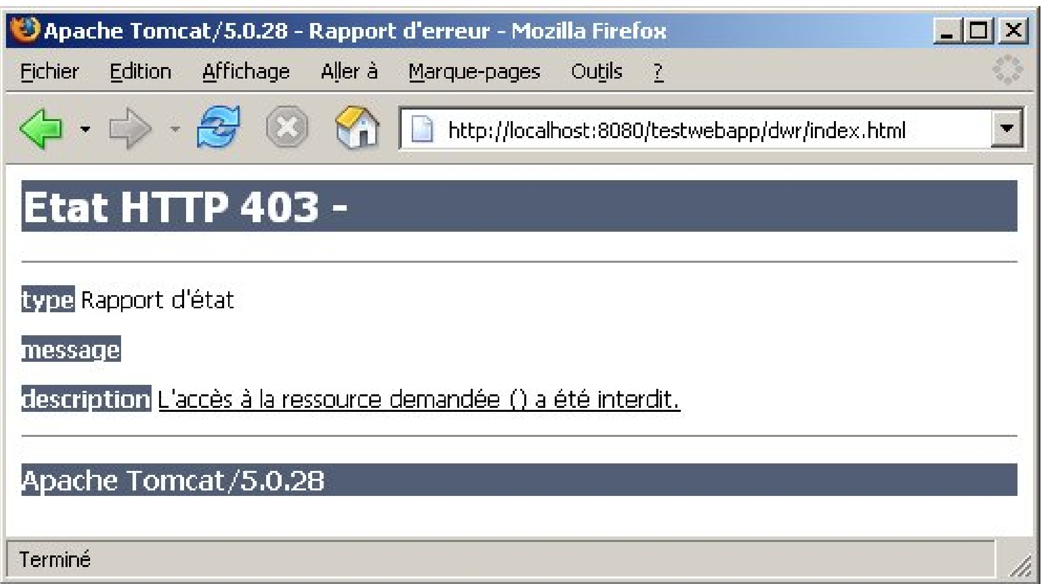
Pour exécuter dynamiquement une méthode sans paramètre, il suffit de simplement cliquer sur le bouton "Execute" de la méthode correspondante.

- equals( ); Execute
  (Warning: No Converter for java.lang.Object. See below)
- toString(); Execute Wed May 10 18:03:35 CEST 2006
- parse(""); Execute

Pour exécuter dynamiquement une méthode avec paramètres, il suffit de saisir leur valeur dans leur zone respective et de cliquer sur le bouton "Execute".

Si la valeur retournée par la méthode n'est pas une valeur simple, alors le résultat est affiché dans une boîte de dialogue.

Si le paramètre debug de la servlet DWR est à false, il n'est pas possible d'accéder à ces fonctionnalités de tests.



Ce mode debug proposé par DWR est particulièrement utile lors de la phase de développement pour vérifier toutes les méthodes qui sont prises en compte par DWR et les tester. Il est cependant fortement déconseillé de le laisser dans un contexte de production pour des raisons de sécurité.

Pour permettre l'utilisation des scripts générés, il suffit de faire un copier/coller dans la partie en-tête de la page HTML des tags <SCRIPT> proposés dans la page de tests de DWR.

Exemple :

```
<script type='text/javascript' src='/testwebapp/dwr/interface/JDate.js'></script>
<script type='text/javascript' src='/testwebapp/dwr/engine.js'></script>
<script type='text/javascript' src='/testwebapp/dwr/util.js'></script>
```

Remarque : il est possible d'utiliser un chemin relatif plutôt qu'un chemin absolu pour ces ressources.

64.4.1.2. Le fichier DWR.xml

Le fichier dwr.xml permet de configurer DWR. Il est généralement placé dans le répertoire WEB-INF de l'application web exécutant DWR.

Le fichier dwr.xml à la structure suivante :

Exemple :

```
<!DOCTYPE dwr PUBLIC
    "-//GetAhead Limited//DTD Direct Web Remoting 1.0//EN"
    "http://www.getahead.ltd.uk/dwr/dwr10.dtd">

<dwr>

  <init>
    <creator id="..." class="..."/>
    <converter id="..." class="..."/>
  </init>

  <allow>
    <create creator="..." javascript="..."/>
    <convert converter="..." match="..."/>
  </allow>

  <signatures>
    ...
  </signatures>

</dwr>
```

Le tag optionnel <init> permet de déclarer ses propres créateurs et convertisseurs. Généralement, ce tag n'est pas utilisé car les créateurs et convertisseurs fournis en standard sont suffisants.

Le tag <allow> permet de définir les objets qui seront utilisés par DWR.

Le tag <create> permet de préciser la façon dont un objet va être instancié. Chaque classe qui pourra être appelée via DWR doit être déclarée avec un tel tag. Ce tag possède la structure suivante :

Exemple :

```
<allow>
  <create creator="..." javascript="..." scope="...">
    <param name="..." value="..."/>
    <auth method="..." role="..."/>
    <exclude method="..."/>
    <include method="..."/>
  </create>
  ...
</allow>
```

Les tags fils <param>, <auth>, <exclude>, <include> sont optionnels

La déclaration d'au moins un créateur est obligatoire. Il existe plusieurs types de créateur spécifiés par l'attribut creator du tag fils <create> :

Type de créateur	Rôle
new	Instancie l'objet avec l'opérateur new
null	Ne créé aucune instance. Ceci est utile si la ou les méthodes utilisées sont statiques
scripted	Instancie l'objet en utilisant un script via BSF
spring	Le framework Spring est responsable de l'instanciation de l'objet
jsf	Utilise des objets de JSF
struts	Utilise des ActionForms de Struts
pageflow	Permet l'action au PageFlow de Beehive ou WebLogic

L'attribut Javascript permet de donner le nom de l'objet Javascript. Il ne faut pas utiliser comme valeur un mot réservé de Javascript.

L'attribut optionnel scope permet de préciser la portée du bean. Les valeurs possibles sont : application, session, request et page. Sa valeur par défaut est page.

Le tag <param> permet de fournir des paramètres au créateur. Par exemple, avec le creator new, il est nécessaire de fournir en paramètre le nom de la classe dont la valeur précise la classe pleinement qualifiée à instancier

Exemple :

```
<!DOCTYPE dwr PUBLIC
    "-//GetAhead Limited//DTD Direct Web Remoting 1.0//EN"
    "http://www.getahead.ltd.uk/dwr/dwr10.dtd">
<dwr>
  <allow>
    <create creator="new" javascript="JDate">
      <param name="class" value="java.util.Date"/>
    </create>
    <create creator="new" javascript="TestDWR">
      <param name="class" value="com.jmd.test.ajax.dwr.TestDWR"/>
    </create>
  </allow>
</dwr>
```

DWR propose un mécanisme via le fichier de configuration dwr.xml qui permet de limiter les méthodes qui seront accessibles via DWR. Les tags <include> et <exclude> permettent respectivement d'autoriser ou d'exclure l'utilisation d'une liste de méthodes. Ces deux tags sont mutuellement exclusifs. En l'absence de l'un de ces deux tags, toutes les méthodes sont utilisables.

Le tag <auth> permet de gérer la sécurité d'accès en utilisant les rôles J2EE de l'application : DWR propose donc la prise en compte des rôles J2EE définis dans le conteneur web pour restreindre l'accès à certaines classes.

Le tag <converter> permet de préciser la façon dont un objet utilisé en paramètre ou en type de retour va être converti. Cette conversion est réalisée par un convertisseur qui assure la transformation des données entre le format des objets client (Javascript) et serveur (Java).

Chaque bean utilisé en tant que paramètre doit être déclaré dans un tel tag. Par défaut, l'utilisation du tag <converter> est inutile pour les primitives, les wrappers de ces primitives (Integer, Float, ...), la classe String, java.util.Date, les tableaux de ces types, les collections (List, Set, Map, ...) et certains objets de manipulation XML issus de DOM, JDOM et DOM4J.

Les convertisseurs Bean et Objet fournis en standard doivent être explicitement utilisés dans le fichier dwr.xml pour des raisons de sécurité.

Exemple :

```
<!DOCTYPE dwr PUBLIC
    "-//GetAhead Limited//DTD Direct Web Remoting 1.0//EN"
    "http://www.getahead.ltd.uk/dwr/dwr10.dtd">
<dwr>
  <allow>
    <create creator="new" javascript="TestDWR">
      <param name="class" value="com.jmd.test.ajax.dwr.TestDWR"/>
    </create>

    <convert converter="bean" match="com.jmd.test.ajax.dwr.Personne"/>
  </allow>
</dwr>
```

Il est possible d'utiliser le caractère joker *

Exemple :

```
    <convert converter="bean" match="com.jmd.test.ajax.dwr.*"/>
    <convert converter="bean" match="*"/>
```

Le convertisseur Bean permet de convertir un Bean en un tableau associatif Javascript et vice et versa en utilisant les mécanismes d'introspection.

Exemple :

```
public class Personne {
  public void setNom(String nom) { ... }
  public void setTaille(int taille) { ... }
  // ...
}
```

L'appel d'une méthode acceptant la classe Personne en paramètre peut se faire de la manière suivante dans la partie cliente :

Exemple :

```
var personne = { nom:"Test", taille:33 };
TestDWR.setPersonne(personne);
```

Il est possible de restreindre l'accès à certaines propriétés d'un bean dans son convertisseur.

Exemple :

```
<convert converter="bean" match="com.jmd.test.ajax.dwr.Personne"/>
  <param name="exclude" value="dateNaissance, taille"/>
</convert>
```

Exemple :

```
<convert converter="bean" match="com.jmd.test.ajax.dwr.Personne"/>
  <param name="include" value="nom, prenom"/>
</convert>
```

L'utilisation de ce dernier exemple est recommandée.

Le convertisseur Objet est similaire mais il utilise directement les membres plutôt que de passer par les getter/setter.

Il possède un paramètre force qui permet d'autoriser l'accès aux membres privés de l'objet par introspection.

Exemple :

```
<convert converter="object" match="com.jmd.test.ajax.dwr.Personne"/>
  <param name="force" value="true"/>
</convert>
```

64.4.1.3. Les scripts engine.js et util.js

Pour utiliser ces deux bibliothèques, il est nécessaire de les déclarer dans chaque page utilisant DWR.

Exemple :

```
<script type='text/javascript' src='/[WEB-APP]/dwr/engine.js'></script>
<script type='text/javascript' src='/[WEB-APP]/dwr/util.js'></script>
```

Le fichier engine.js est la partie principale côté Javascript puisqu'il assure toute la gestion de la communication avec le serveur.

Certaines options de paramétrage peuvent être configurées en utilisant la fonction DWREngine.setX().

Il est possible de regrouper plusieurs communications en une seule en utilisant les fonctions DWREngine.beginBatch() et DWREngine.endBatch(). Lors de l'appel de cette dernière, les appels sont réalisés vers le serveur. Ce regroupement permet de réduire le nombre d'objets XMLHttpRequest créés et le nombre de requêtes envoyées au serveur.

Le fichier util.js propose des fonctions utilitaires pour faciliter la mise à jour dynamique de la page. Ces fonctions ne sont pas dépendantes d'autres éléments de DWR.

Fonction	Rôle
$(id)	Encapsuler un appel à la fonction document.getElementById() comme dans la bibliothèque Prototype
addOptions	Ajouter des éléments dans une liste ou un tag ou
removeAllOptions	Supprimer tous les éléments d'une liste ou un tag ou
addRows	Ajouter une ligne dans un tableau
removeAllRows	Supprimer toutes les lignes dans un tableau
getText	Renvoyer la valeur sélectionnée dans une liste
getValue	Renvoyer la valeur d'un élément HTML
getValues	Obtenir les valeurs de plusieurs éléments fournis sous la forme d'un ensemble de paires clé:valeur_vide dont la clé est l'id de l'élément à traiter
onReturn	Gérer l'appui sur la touche return avec un support multi-navigateur
selectRange(id, debut,fin)	Gérer une sélection dans une zone de texte avec un support multi-navigateur
setValue(id,value)	Mettre à jour la valeur d'un élément
setValues	Mettre à jour les valeurs de plusieurs éléments fournis sous la forme d'un ensemble de paires clé:valeur dont la clé est l'id de l'élément à modifier
toDescriptiveString(id, level)	Afficher des informations sur un objet avec un niveau de détail (0, 1ou 2)
useLoadingMessage	Mettre en place un message de chargement lors des échanges avec le serveur

64.4.1.4. Les scripts client générés

DWR assure un mapping entre les méthodes des objets Java et les fonctions Javascript générées. Chaque objet Java est mappé sur un objet Javascript dont le nom correspond à la valeur de l'attribut javascript du creator correspondant dans le fichier de configuration de DWR.

Le nom des méthodes est conservé comme nom de fonction dans le code Javascript. Le premier paramètre de toutes les fonctions générées par DWR est la fonction de type callback qui sera exécutée à la réception de la réponse. Les éventuels autres paramètres correspondant à leur équivalent dans le code Java.

DWR s'occupe de transformer un objet Java en paramètre ou en résultat en un équivalent dans le code Javascript. Par exemple, une collection Java est transformée en un tableau d'objets Javascript de façon transparente. Ceci rend transparent la conversion et facilite donc leur utilisation.

L'utilisation de la bibliothèque util.js peut être particulièrement pratique pour faciliter l'exploitation des données retournées et utilisées par les fonctions générées.

Des exemples d'utilisation sont fournis dans les sections d'exemples suivantes.

64.4.1.5. Un exemple pour obtenir le contenu d'une page

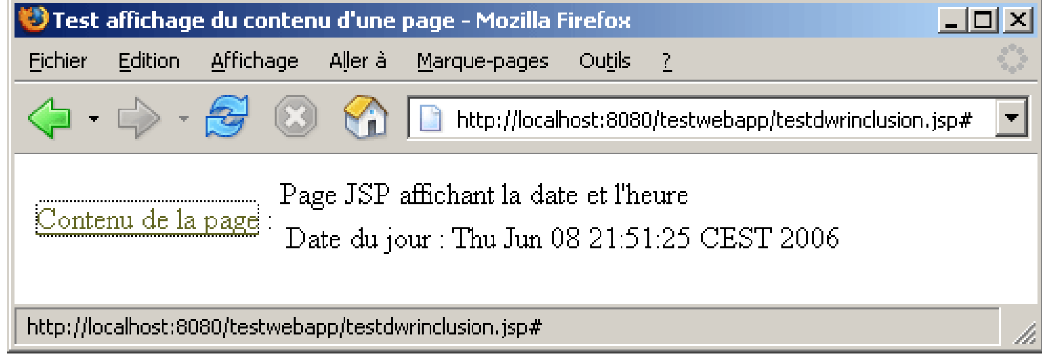
Il est possible qu'une méthode d'un bean renvoie le contenu d'une JSP en utilisant l'objet uk.ltd.getahead.dwr.ExecutionContext. Cet objet permet d'obtenir le contenu d'une url donnée.

Exemple : la JSP dont le contenu sera retourné

```
Page JSP affichant la date et l'heure
<table>
  <tr>
    <td>Date du jour :</td>
    <td nowrap><%=new java.util.Date()%></td>
  </tr>
</table>
```

Exemple :

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
  pageEncoding="ISO-8859-1"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Test affichage du contenu d'une page</title>
<script type='text/javascript'
  src='/testwebapp/dwr/interface/TestDWR.js'></script>
<script type='text/javascript' src='/testwebapp/dwr/engine.js'></script>
<script type='text/javascript' src='/testwebapp/dwr/util.js'></script>

<script type='text/javascript'>
<!--

function inclusion() {
  TestDWR.getContenuPage(afficherInclusion);
}

function afficherInclusion(data) {
  DWRUtil.setValue("inclusion", data);
}

function init() {
  DWRUtil.useLoadingMessage();
}

-->
</script>
</head>
<body onload="init();">

<table>
  <tr>
    <td><a href="#" onclick="inclusion()">Contenu de la page</a> :</td>
    <td nowrap>
    <div id="inclusion"></div>
    </td>
  </tr>
</table>

</body>
</html>
```



Lors d'un clic sur le lien, le contenu de la JSP est affiché dans le calque.

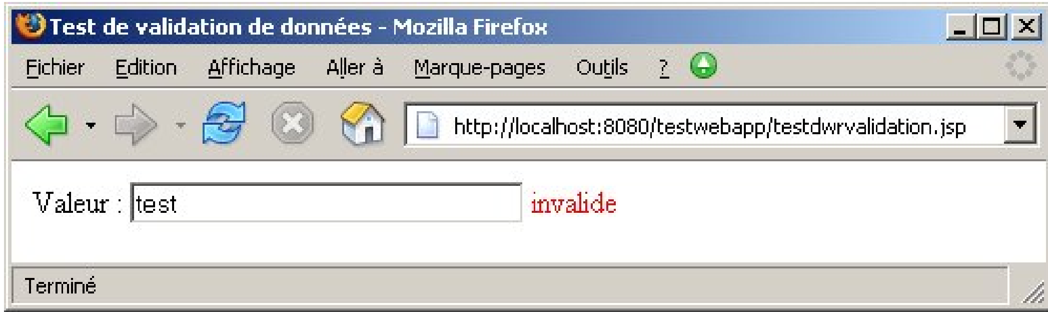
64.4.1.6. Un exemple pour valider des données

Dans cet exemple, à chaque saisie dans la zone de texte, le contenu est validé à la volée par un appel à une méthode d'un bean.

Exemple :

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
        pageEncoding="ISO-8859-1"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Test de validation de données</title>
<script type='text/javascript' src='/testwebapp/dwr/interface/TestDWR.js'></script>
<script type='text/javascript' src='/testwebapp/dwr/engine.js'></script>
<script type='text/javascript' src='/testwebapp/dwr/util.js'></script>

<script type='text/javascript'>
<!--
function valider() {
  TestDWR.validerValeur(afficherValidation, $("donnees").value);
}

function afficherValidation(data) {
  DWRUtil.setValue("validationMessage",data);
  if (data == "valide") {
    $("validationMessage").style.color='#00FF00';
  } else {
    $("validationMessage").style.color='#FF0000';
  }
}

function init() {
  DWRUtil.useLoadingMessage();
}
-->
</script>
</head>
<body onload="init();">

<table>
        <tr>
                <td>Valeur :</td>
                <td nowrap><input type="text" id="donnees" name="donnees" size="30"
                        onkeyup="valider();"></td>
                <td>
                <div id="validationMessage"></div>
                </td>
        </tr>
</table>

</body>
```

```
</html>
```

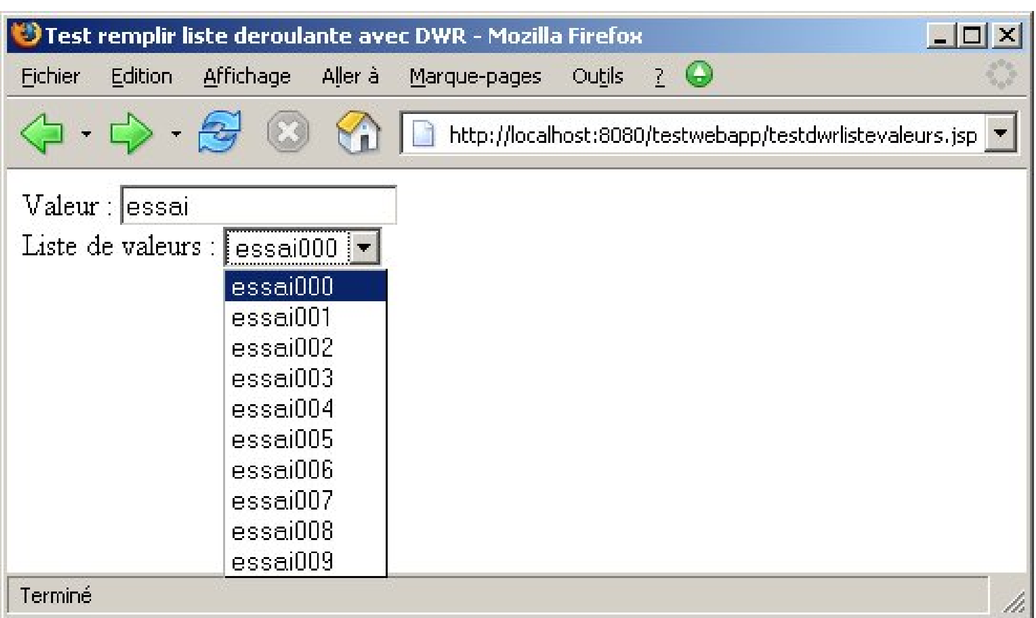


Exemple :

```
  public String validerValeur(String valeur) {

    String resultat = "invalide";

    if ((valeur != null) && valeur.startsWith("X")) {
      resultat = "valide";
    }

    return resultat;
  }
```

64.4.1.7. Un exemple pour remplir dynamiquement une liste déroulante

Cet exemple va remplir dynamiquement le contenu d'une liste déroulante en fonction de la valeur d'une zone de saisie.



Côté serveur la méthode getListeValeurs() du bean est appelée pour obtenir les valeurs de la liste déroulante. Elle attend en paramètre une chaîne de caractères et renvoie un tableau de chaînes de caractères.

Exemple :

```
package com.jmd.test.ajax.dwr;

public class TestDWR {

  public String[] getListeValeurs(String valeur)
```

```
  {
    String[] resultat = new String[10];

    for(int i = 0 ; i <10;i++ ) {
      resultat[i] = valeur+"00"+i;
    }

    return resultat;
  }

}
```

La page de l'application est composée d'une zone de saisie et d'une liste déroulante.

Exemple :

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
        pageEncoding="ISO-8859-1"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Test remplir liste deroulante avec DWR</title>
<script type='text/javascript' src='/testwebapp/dwr/interface/TestDWR.js'></script>
<script type='text/javascript' src='/testwebapp/dwr/engine.js'></script>
<script type='text/javascript' src='/testwebapp/dwr/util.js'></script>

<script type='text/javascript'>
<!--
function rafraichirListeValeurs() {
  TestDWR.getListeValeurs(remplirListeValeurs, $("valeur").value);
}

function remplirListeValeurs(data) {
  DWRUtil.removeAllOptions("listevaleurs");
  DWRUtil.addOptions("listevaleurs", data);
  DWRUtil._selectListItem($("listevaleurs"),$("listevaleurs").options[0].value);
}

function init() {
  DWRUtil.useLoadingMessage();
  rafraichirListeValeurs();
}
-->
</script>
</head>
<body onload="init();">

<p>Valeur : <input type="text" id="valeur"
        onblur="rafraichirListeValeurs();" /><br />
Liste de valeurs : <select id="listevaleurs" style="vertical-align:top;"></select>
</p>

</body>
</html>
```

La fonction init() se charge d'initialiser le contenu de la liste déroulante au chargement de la page.

La fonction rafraichirListeValeurs() est appelée dès que la zone de saisie perd le focus. Elle utilise la fonction Javascript TestDWR.getListeValeurs() générée par DWR pour appeler la méthode du même nom du bean. Les deux paramètres fournis à cette fonction permettent de préciser que c'est la fonction remplirListeValeurs() qui fait office de fonction de callback et de fournir la valeur de la zone de saisie en paramètre de l'appel de la méthode getListeValeurs() du bean.

La fonction remplirListeValeurs() se charge de vider la liste déroulante, de remplir son contenu avec les données reçues en réponse du serveur (elles sont passées en paramètre de la fonction) et de sélectionner le premier élément de la liste. Pour ces trois actions, trois fonctions issues de la bibliothèque util.js de DWR sont utilisées.

La fonction addOptions() utilise les données passées en paramètre pour remplir la liste.

64.4.1.8. Un exemple pour afficher dynamiquement des informations

L'exemple de cette section va permettre d'afficher dynamiquement les données d'une personne sélectionnée. L'exemple est volontairement simpliste (la liste déroulante des personnes est en dur et les données de la personne sont calculées plutôt qu'extraite d'une base de données). Le but principal de cet exemple est de montrer la facilité d'utilisation des beans mappés par DWR dans le code Javascript.

Le bean utilisé encapsule les données d'une personne

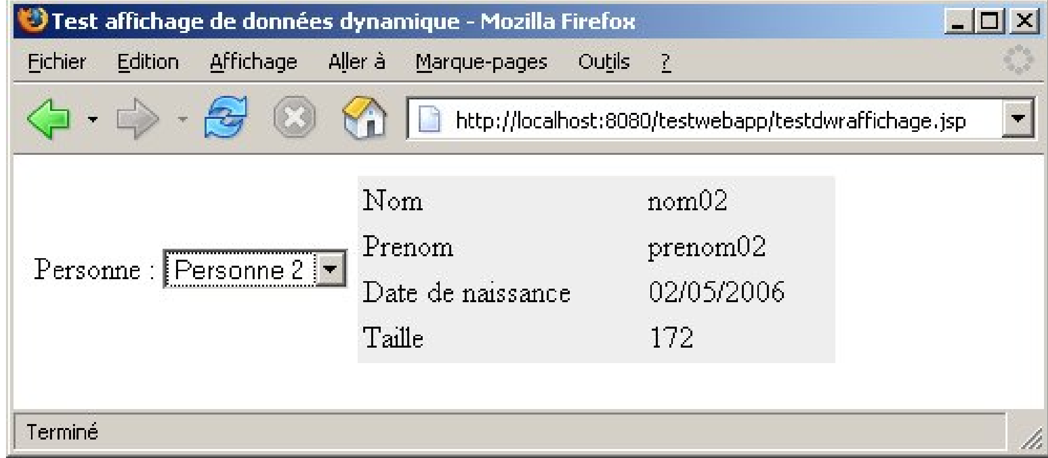
Exemple :

```
package com.jmd.test.ajax.dwr;

import java.util.Date;

public class Personne {
  private String nom;
  private String prenom;
  private String dateNaissance;
  private int taille;

  public Personne() {
    super();
  }

  public Personne(String nom, String prenom, String dateNaissance, int taille) {
    super();
    this.nom = nom;
    this.prenom = prenom;
    this.dateNaissance = dateNaissance;
    this.taille = taille;
  }

  public String getDateNaissance() {
    return dateNaissance;
  }

  public void setDateNaissance(String dateNaissance) {
    this.dateNaissance = dateNaissance;
  }

  public String getNom() {
    return nom;
  }

  public void setNom(String nom) {
    this.nom = nom;
  }

  public String getPrenom() {
    return prenom;
  }

  public void setPrenom(String prenom) {
    this.prenom = prenom;
  }

  public int getTaille() {
    return taille;
  }

  public void setTaille(int taille) {
    this.taille = taille;
  }

}
```

La page est composée d'une liste déroulante de personnes. Lorsqu'une personne est sélectionnée, les données de cette personne sont demandées au serveur et sont affichées.

Exemple :

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
        pageEncoding="ISO-8859-1"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Test affichage de données dynamique</title>
<script type='text/javascript' src='/testwebapp/dwr/interface/TestDWR.js'></script>
<script type='text/javascript' src='/testwebapp/dwr/engine.js'></script>
<script type='text/javascript' src='/testwebapp/dwr/util.js'></script>

<script type='text/javascript'>
<!--
function rafraichir() {
  TestDWR.getPersonne(afficherPersonne, $("personnes").value);
}

function afficherPersonne(data) {
  DWRUtil.setValue("nomPersonne",data.nom);
  DWRUtil.setValue("prenomPersonne",data.prenom);
  DWRUtil.setValue("datenaissPersonne",data.dateNaissance);
  DWRUtil.setValue("taillePersonne",data.taille);
}

function init() {
  DWRUtil.useLoadingMessage();
}
-->
</script>
</head>
<body onload="init();">

<table>
        <tr>
                <td>Personne :</td>
                <td nowrap><select id="personnes" name="personnes"
                        onchange="rafraichir();">
                        <option value="1">Personne 1</option>
                        <option value="2">Personne 2</option>
                        <option value="3">Personne 3</option>
                        <option value="4">Personne 4</option>
                        </select>
                </td>
                <td>
                <div id="informationPersonne">
                <table bgcolor="#eeeeee" width="250">
                <tr><td>Nom</td><td><span id="nomPersonne"></span></td></tr>
                <tr><td>Prenom</td><td><span id="prenomPersonne"></span></td></tr>
                <tr><td>Date de naissance</td><td><span id="datenaissPersonne"></span></td></tr>
                <tr><td>Taille</td><td><span id="taillePersonne"></span></td></tr>
                </table>
                </div>
                </td>
        </tr>
</table>

</body>
</html>
```



Exemple : le source de la méthode du bean qui recherche les données de la personne

```
public Personne getPersonne(String id) {
  int valeur = Integer.parseInt(id);
  if (valeur < 10) {
    id = "0"+id;
  }
  Personne resultat = new Personne("nom"+id,"prenom"+id,id+"/05/2006",170+valeur);
  return resultat;
}
```

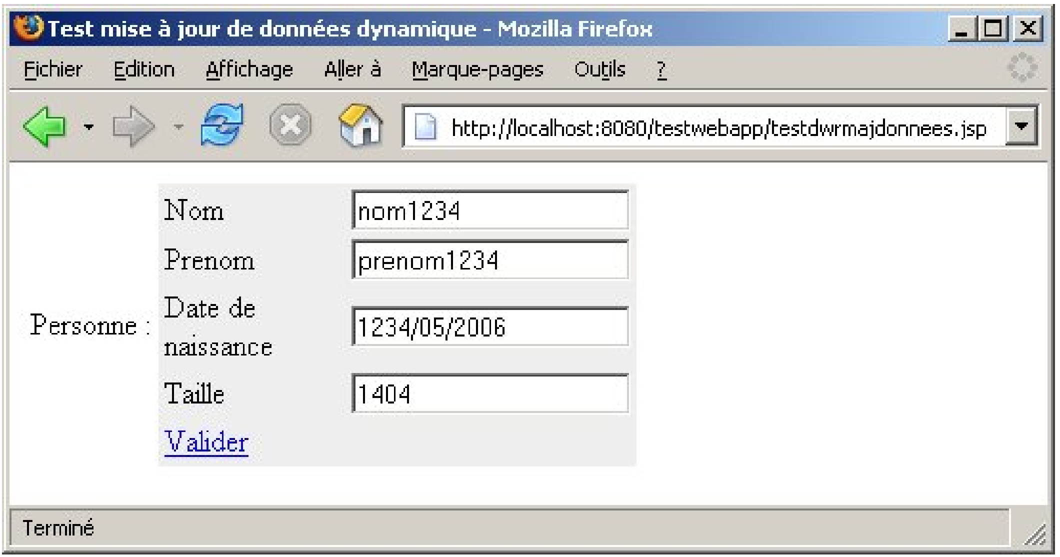
Dans le fichier de configuration dwr.xml, un convertisseur de type bean doit être déclaré pour le bean de type Personne

Exemple :

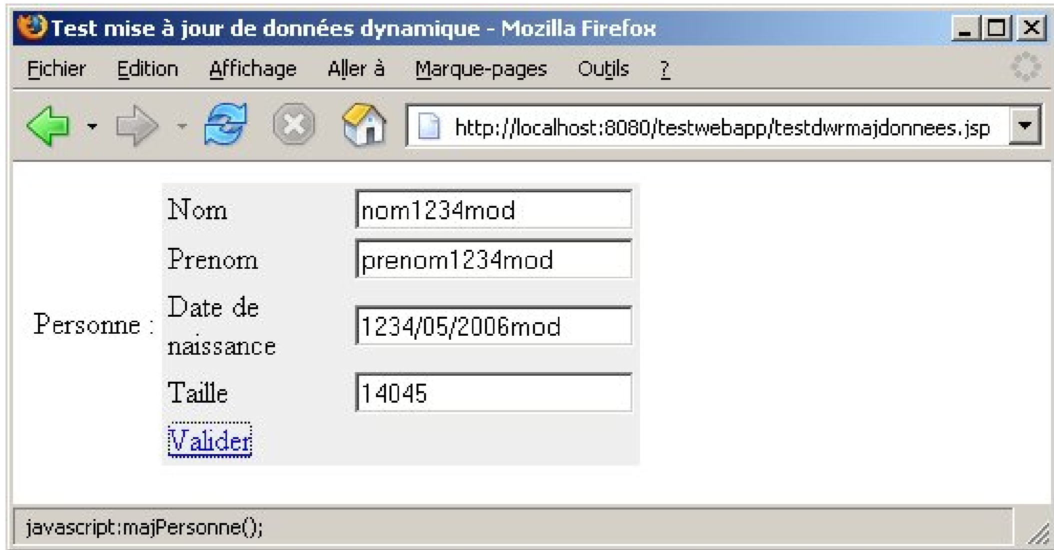
```
<allow>
  <create creator="new" javascript="TestDWR">
    <param name="class" value="com.jmd.test.ajax.dwr.TestDWR"/>
  </create>

  <convert converter="bean" match="com.jmd.test.ajax.dwr.Personne"/>
</allow>
```

64.4.1.9. Un exemple pour mettre à jour des données

Cet exemple va permettre de modifier les données d'une personne.



Il suffit de modifier les données et de cliquer sur le bouton valider



Les données sont envoyées sur le serveur.

Exemple :

```
INFO: Exec[0]: TestDWR.setPersonne()
nom=nom1234mod
prenom=prenom1234mod
datenaiss=1234/05/2006mod
taille14045
```

Exemple : le source de la méthode du bean qui recherche les données de la personne

```
  public void setPersonne(Personne personne)
  {
    System.out.println("nom="+personne.getNom());
    System.out.println("prenom="+personne.getPrenom());
    System.out.println("datenaiss="+personne.getDateNaissance());
    System.out.println("taille"+personne.getTaille());
      // code pour rendre persistant l'objet fourni en paramètre
  }
```

Cette méthode affiche simplement les données reçues. Dans un contexte réel, elle assurerait les traitements pour rendre persistantes les modifications dans les données reçues.

La page de l'application est la suivante.

Exemple :

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
        pageEncoding="ISO-8859-1"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Test mise à jour de données dynamique</title>
<script type='text/javascript' src='/testwebapp/dwr/interface/TestDWR.js'></script>
<script type='text/javascript' src='/testwebapp/dwr/engine.js'></script>
<script type='text/javascript' src='/testwebapp/dwr/util.js'></script>

<script type='text/javascript'>
<!--
var personne;

function rafraichir() {
  TestDWR.getPersonne(afficherPersonne, "1234");
```

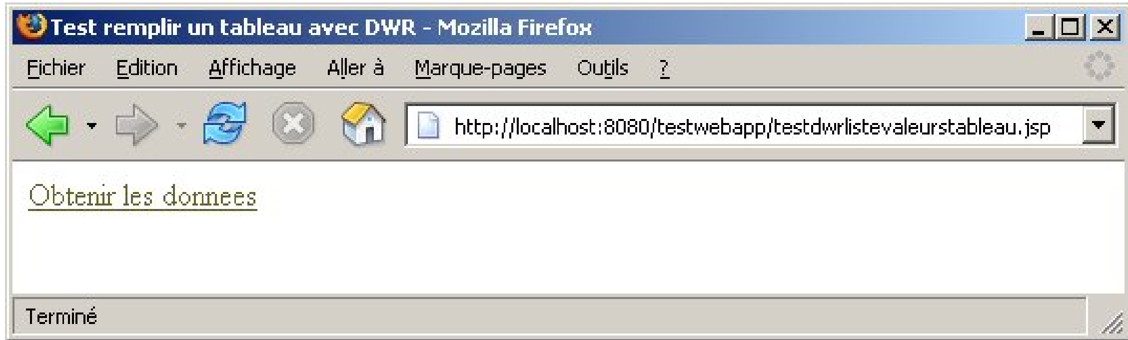
```
}

function afficherPersonne(data) {
  personne = data;
  DWRUtil.setValues(data);
}

function majPersonne()
{
    DWRUtil.getValues(personne);
    TestDWR.setPersonne(personne);
}

function init() {
  DWRUtil.useLoadingMessage();
  rafraichir();
}

-->
</script>
</head>
<body onload="init();">

<table>
  <tr>
    <td>Personne :</td>
    <td>
    <div id="informationPersonne">
    <table bgcolor="#eeeeee" width="250">
    <tr><td>Nom</td><td><input type="text" id="nom"></td></tr>
    <tr><td>Prenom</td><td><input type="text" id="prenom"></td></tr>
    <tr><td>Date de naissance</td><td><input type="text" id="dateNaissance"></td></tr>
    <tr><td>Taille</td><td><input type="text" id="taille"></td></tr>
    <tr><td colspan="2"><a href="javascript:majPersonne();">Valider</a></td></tr>
    </table>
    </div>
    </td>
        </tr>
</table>

</body>
</html>
```

Cet exemple utilise les fonctions getValues() et setValues() qui mappent automatiquement les propriétés d'un objet avec les objets de l'arbre DOM dont l'id correspond.

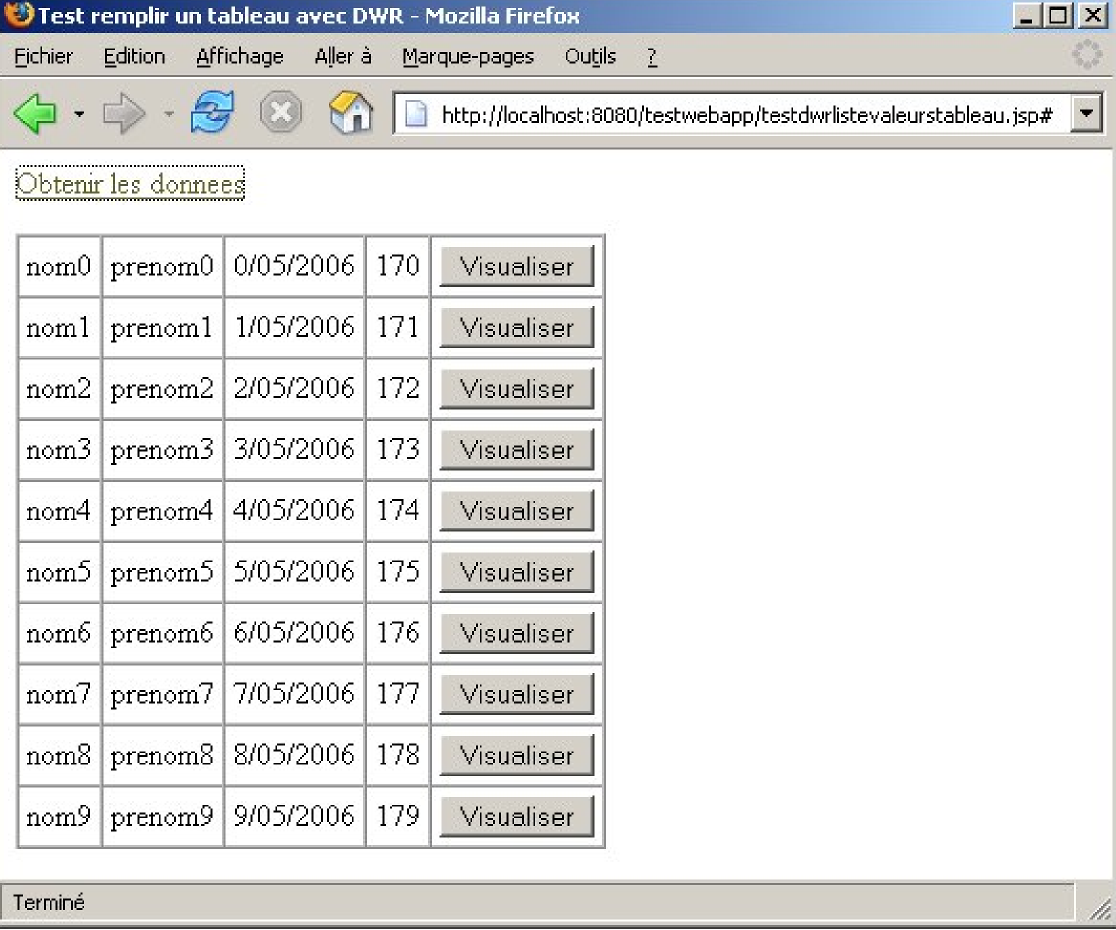
Remarque : il est important que l'objet personne qui encapsule les données de la personne soit correctement initialisé, ce qui est fait au chargement des données de la personne.

64.4.1.10. Un exemple pour remplir dynamiquement un tableau de données

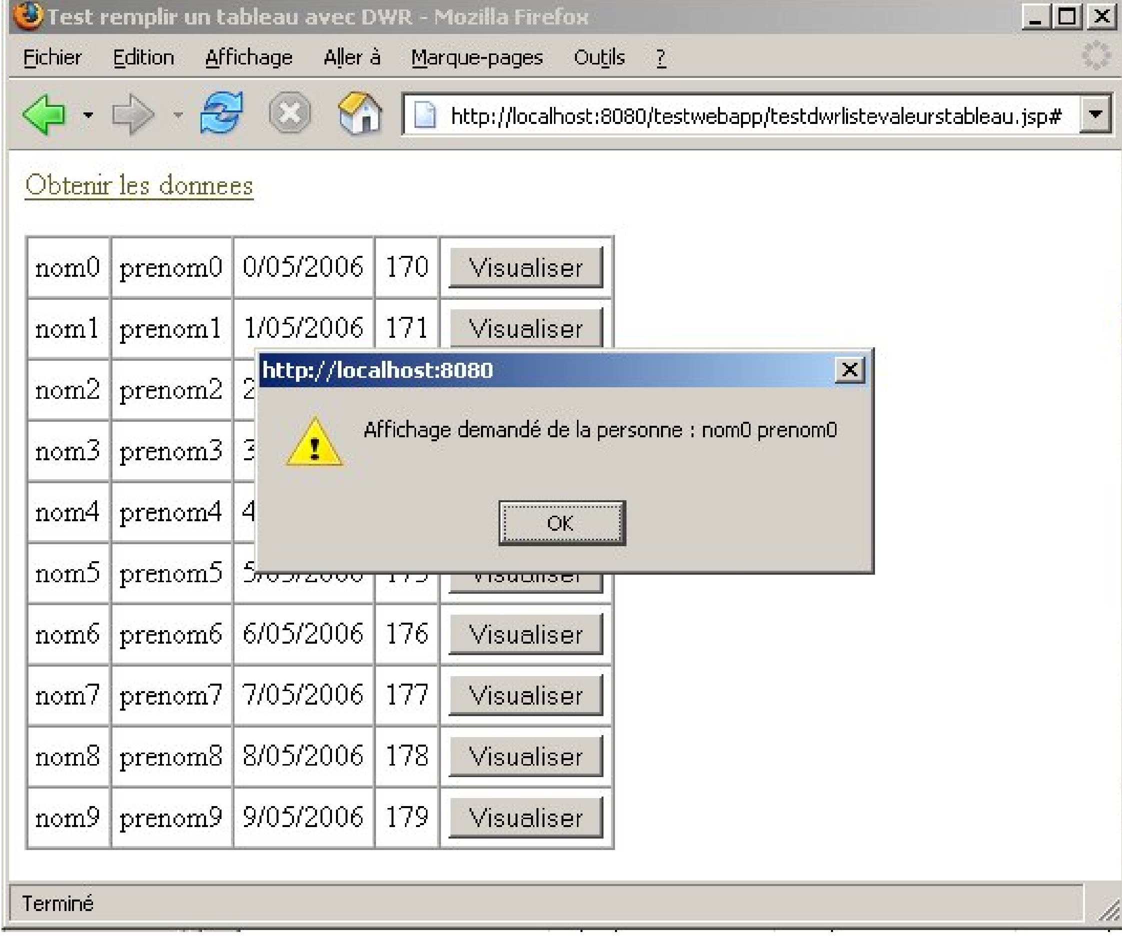
Cet exemple va remplir dynamiquement le contenu d'un tableau avec une collection d'objets.



Un clic sur le lien permet d'afficher le tableau avec les données retournées par le serveur.



Un clic sur le bouton "visualiser" affiche un message avec le nom et la personne concernée.



Côté serveur la méthode getPersonnes() du bean est appelée pour obtenir la liste des personnes sous la forme d'une collection d'objets de type Personne.

Exemple :

```
  public List getPersonnes() {
    List resultat = new ArrayList();
    Personne personne = null;

    for (int i = 0; i<10 ; i++) {
      personne = new Personne("nom"+i,"prenom"+i,i+"/05/2006",170+i);
      resultat.add(personne);
    }
    return resultat;
  }
```

La page de l'application est composée d'un calque contenant un tableau.

Exemple :

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
        pageEncoding="ISO-8859-1"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Test remplir un tableau avec DWR</title>
<script type='text/javascript' src='/testwebapp/dwr/interface/TestDWR.js'></script>
<script type='text/javascript' src='/testwebapp/dwr/engine.js'></script>
<script type='text/javascript' src='/testwebapp/dwr/util.js'></script>

<script type='text/javascript'>
<!--
function rafraichirListeValeurs() {
  TestDWR.getPersonnes(remplirListeValeurs);
}

function remplirListeValeurs(data) {

  DWRUtil.removeAllRows("tableau");

  if (data.length == 0) {
    alert("");
    $("donnees").style.visibility = "hidden";
  } else {
    DWRUtil.addRows("tableau",data,cellulesFonctions);
    $("donnees").style.visibility = "visible";
  }
}

// tableau des fonctions permettant d'assurer le rendu des différentes cellules du tableau
var cellulesFonctions = [
  function(item) { return item.nom; },
  function(item) { return item.prenom; },
  function(item) { return item.dateNaissance; },
  function(item) { return item.taille; },
  function(item) {
    var btn = document.createElement("button");
    btn.innerHTML = "Visualiser";
    btn.itemId = item.nom+" "+item.prenom;
    btn.onclick = afficherPersonne;
    return btn;
  }
];


function afficherPersonne() {
  alert("Affichage demandé de la personne : "+this.itemId);
}

function init() {
```

```
  DWRUtil.useLoadingMessage();
}
-->
</script>
</head>
<body onload="init();">

<p><a href="#" onclick="rafraichirListeValeurs()">Obtenir les donnees</a></p>
<div id="donnees">
  <table id="tableau" border="1" cellpadding="4" cellspacing="0"></table>
</div>
</body>
</html>
```

Cet exemple met en oeuvre les fonctions de manipulation de tableau de la bibliothèque util.js notamment la fonction DWRUtil.addRows("tableau",data,cellulesFonctions) qui permet d'ajouter un ensemble de lignes à un tableau HTML.

Elle attend en paramètre l'id du tableau à modifier, les données à utiliser et un tableau de fonctions qui vont définir le rendu de chaque cellule d'une ligne du tableau. Ces fonctions peuvent simplement retourner la valeur d'une propriété de l'objet courant ou renvoyer des objets plus complexes comme un bouton.

Remarque : il est préférable d'utiliser un id pour les boutons générés qui soit plus adapté pour une meilleure unicité. Il serait souhaitable d'utiliser un suffixe et un identifiant unique plutôt que la concaténation du nom et du prénom. Ce choix, dans cet exemple, a été fait pour le conserver le plus simple possible.

64.4.2. D'autres frameworks

Ils existent de nombreux autres frameworks permettant de mettre en oeuvre Ajax dont voici une liste non exhaustive :

Framework	Url
AjaxAnywhere	http://ajaxanywhere.sourceforge.net
AjaxTags	http://ajaxtags.sourceforge.net/
ajax4suggest	http://sourceforge.net/projects/ajax4suggest
AJAX-JSF	http://smirnov.org.ru/en/ajax-jsf.html
Echo 2	http://www.nextapp.com/products/echo2/
SWATO	http://swato.dev.java.net/
WidgetServer	http://wiser.dev.java.net
jWic	http://www.jwic.de

65. GWT (Google Web Toolkit)

GWT (Google Web Toolkit) est un framework open source de développement d'applications web mettant en oeuvre AJAX développé par Bruce Johnson et Google.

Mi 2006, Google a diffusé GWT qui est un outil de développement d'applications de type RIA offrant une mise en oeuvre novatrice : le but est de faciliter le développement d'applications web mettant en oeuvre Ajax en faisant abstraction des incompatibilités des principaux navigateurs.

GWT propose de nombreuses fonctionnalités pour développer une application exécutable dans un navigateur avec des comportements similaires à ceux d'une application desktop :

- création d'applications graphiques s'exécutant dans un navigateur
- pas besoin d'écrire du code Javascript sauf pour des besoins très spécifiques comme l'intégration d'une bibliothèque Javascript existante
- utilisation de CSS pour personnaliser l'apparence
- mise en oeuvre d'Ajax sans manipuler l'arbre DOM de la page mais en utilisant des objets Java
- un ensemble riche de composants (widgets et panels)
- communication avec le serveur grâce à des appels asynchrones en échangeant des objets Java et en utilisant des exceptions pour signifier des problèmes
- internationalisation
- un système de gestion de l'historique sur le navigateur
- un parser XML
- détection des erreurs à la compilation
- ...

L'utilisation de GWT possède plusieurs avantages :

- pas de code Javascript à écrire
- utilisation de Java comme langage de développement
- une meilleure productivité liée à l'utilisation uniquement de Java (un seul langage à utiliser, mieux connu que d'autres technologies notamment Javascript, mise en oeuvre d'un débogueur, utilisation d'un IDE Java, ...)
- hormis les styles CSS et la page HTML qui encapsule l'application il n'y a pas d'utilisation directe de technologies web
- le code généré par GWT supporte les principaux navigateurs
- la prise en main est facile même pour des débutants ce qui lui confert une bonne courbe d'apprentissage

Le site officiel de GWT est à l'url http://code.google.com/webtoolkit/

Ce chapitre contient plusieurs sections :

- La présentation de GWT
- La création d'une application
- Les modes d'exécution
- Les éléments de GWT
- L'interface graphique des applications GWT
- La personnalisation de l'interface
- Les composants (widgets)
- Les panneaux (panels)
- La création d'éléments réutilisables

♦ Les événements
♦ JSNI
♦ La configuration et l'internationalisation
♦ L'appel de procédures distantes (Remote Procedure Call)
♦ La manipulation des documents XML
♦ La gestion de l'historique sur le navigateur
♦ Les tests unitaires
♦ Le déploiement d'une application
♦ Des composants tiers
♦ Les ressources relatives à GWT

65.1. La présentation de GWT

Le code de l'application est entièrement écrit en Java notamment la partie cliente qui devra s'exécuter dans un navigateur. Ce code Java n'est pas compilé en byte code mais en Javascript ce qui permet son exécution dans un navigateur.

Le coeur de GWT est donc composé du compilateur de code Java en Javascript. L'avantage du code Javascript produit est qu'il est capable de s'exécuter sur les principaux navigateurs sans avoir à en tenir compte dans le code écrit puisque le compilateur créé un fichier Javascript optimisé pour chacun de ces navigateurs.

Le code à écrire pour la partie cliente est composé de plusieurs éléments :

- la syntaxe est celle de Java 1.4 (les fonctionnalités de Java 5 sont supportées à partir de la version 1.5 de GWT)
- un sous ensemble des API de bases du JDK notamment des packages java.lang et java.util (particulièrement celles qui pourront être compilées en JavaScript. La liste complète de ces classes est consultable à l'url http://code.google.com/webtoolkit/documentation/jre.html
- un ensemble de composants graphiques nommés widgets et de panels qui sont utilisés pour réaliser l'interface graphique

La partie graphique d'une application GWT est composée d'une petite partie en HTML, de CSS et surtout de classes Java dans lesquelles des composants sont utilisés avec des gestionnaires d'événements pour définir l'interface de l'application et les réponses aux actions des utilisateurs.

GWT propose un ensemble assez complet de composants graphiques nommés widgets fournis en standard : l'ensemble des widgets inclus des composants graphiques standards (boutons, zone de saisie de texte, liste déroulante, ...) mais contient aussi des composants plus riche tels que des panneaux déroulants, des onglets, des arbres, des boîtes de dialogue, ... Il est aussi possible de créer ces propres composants ou d'intégrer des frameworks Javascript (ext, Dojo, Rialto, Yahoo UIL, ...)

Le code Java pour coder en GWT est très ressemblant à celui à produire pour développer une application graphique utilisant AWT :

- instancier des composants
- ajouter ces composants dans la hiérarchie des composants de la page
- utiliser des gestionnaires d'événements pour répondre aux actions des utilisateurs

Exemple :

```
public class TestBonjour implements EntryPoint {
    public void onModuleLoad() {
        Button bouton = new Button("Saluer", new ClickListener() {
            public void onClick(Widget sender) {
                Window.alert("Bonjour");
            }
        });
        RootPanel.get().add(bouton);
    }
}
```

GWT propose aussi des outils pour assurer la communication avec la partie serveur en reposant sur AJAX et offre aussi un support de JUnit.

Ce framework propose plusieurs originalités intéressantes :

- développement majoritairement en Java et un peu d'HTML : le code Javascript est généré par le compilateur GWT
- développé en Java une application GWT est plus facile à déboguer en utilisant un IDE Java
- GWT fournit de nombreux composants (widgets) et un système de gestion de leur positionnement (Layout)
- l'application GWT gère le support des principaux navigateurs
- ...

Le grand avantage de GWT est que l'application web utilisant Ajax développée avec ce framework ne nécessite essentiellement que des connaissances en Java : quelques rudiments d'HTML et CSS sont nécessaires pour développer des applications GWT mais aucun code Javascript n'est à écrire.

Une application GWT est développée en Java avec un des IDE Java : ceci rend l'application facilement débogable avec les fonctionnalités de l'IDE.

La version 1.4 de GWT repose sur Java 1.4.

La version 1.5 de GWT repose sur Java 1.5 et offre un support des fonctionnalités de Java 5 (énumérations, generics, ...)

Une application GWT est contenue dans un module. Un module est un ensemble de classes et un fichier de configuration. Un module possède un point d'entrée (entry point) qui correspond à la classe principale qui sera utilisée au lancement de l'application.

Côté serveur, il est possible d'utiliser toutes les technologies capables de traiter des requêtes http (Java, .Net, PHP, ...). Java est particulièrement bien adapté à cette tâche grâce à ces nombreuses API et frameworks disponibles.

Une application de gestion développée avec GWT est donc composée de classes Java :

- pour la partie IHM : ces classes Java sont compilées en Javascript pour permettre l'exécution de l'application dans un navigateur
- pour la partie serveur : ces classes assurent les traitements métiers et la persistance de données.

Remarque : la partie serveur n'a pas d'obligation à être développée en Java. Elle peut être développée avec d'autres plateformes mais GWT offre des facilités pour l'utilisation de Java

Lors du déploiement, l'application web sera générée à partir du code Java pour produire la code HTML et Javascript requis pour la partie cliente.

Une application GWT peut être exécutée dans deux modes :

- mode hôte (hosted mode) : il est utilisé lors de la phase de développement. Dans ce mode, l'application est exécutée sous la forme de byte code dans une JVM. Ce mode est donc préconisé durant la phase de développement puisque permet la mise en oeuvre d'un débogueur pour faciliter la mise au point de l'application. Ce mode utilise une version personnalisée d'un navigateur fourni par GWT qui utilise un navigateur selon l'OS (Internet Exploreur sous Windows et Firefox sous Linux) et une machine virtuelle Java qui permet de transformer le code Java et l'afficher dans le navigateur.
- mode web (web mode) : dans ce mode, le code Java est compilé pour générer le code HTML et Javascript de la partie client. L'application peut ainsi être exécutée dans les navigateurs supportés par GWT.

Le mode hôte est un environnement d'exécution fourni par GWT qui est particulièrement adapté aux tests et au débogage. Dans ce mode, le code Java n'est pas compilé en Javascript mais compilé en byte code et exécuté dans une JVM. Il est ainsi possible d'utiliser un débogueur.

65.1.1. L'installation de GWT

Il faut télécharger GWT à l'url : http://code.google.com/webtoolkit/download.html

La version utilisée dans ce chapitre est la 1.3.3 sous Windows. Le fichier téléchargé se nomme donc gwt-windows-1.3.3.zip. Il faut décompresser le contenu de ce fichier dans un répertoire du système en utilisant un outil gérant le format zip comme l'utilitaire jar fourni avec le JDK.

65.1.2. GWT version 1.6

La version 1.6 de GWT est publiée en mai 2006.

Cette version apporte de nombreuses évolutions notamment :

- une nouvelle structure pour les projets qui facilite le packaging de la partie serveur sous la forme d'une archive war
- un nouvel environnement d'exécution local qui utilise Jetty
- de nouveaux mécanismes de gestion des événements et le support d'évènements natifs
- de nouveaux composants (DatePicket, DateBox, LazyPanel)
- corrections de bugs
- ...

65.1.2.1. La nouvelle structure pour les projets

La structure des répertoires des projets GWT a été adaptée notamment pour respecter le standard des applications web pour faciliter la création du packaging de déploiement sous la forme d'une archive war.

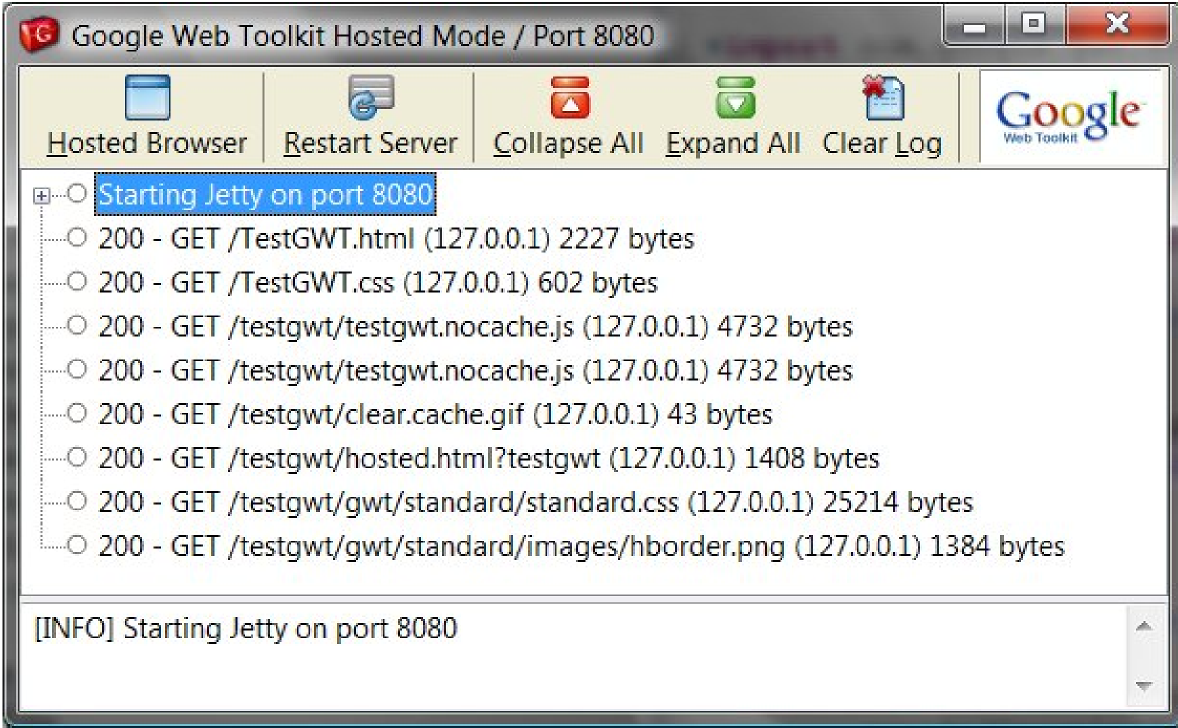
Le répertoire cible de génération des livrables sur nomme d'ailleurs /war : ce répertoire contiendra le résultat des compilations mais aussi les ressources statiques nécessaires aux modules de l'application. Ainsi certaines adaptations sont nécessaires par rapport à l'organisation des projets des versions antérieures notamment :

- Le fichier de configuration web.xml de la partie serveur se trouve directement dans le sous répertoire /war/WEB-INF.
- Toutes les bibliothèques requises par la partie serveur pour traiter les appels GWT-RPC doivent être ajoutées dans le sous répertoire /war/WEB-INF/lib
- La page HTML qui contient l'application ainsi que les ressources requises doivent être mises dans le sous répertoire war et non plus dans le sous répertoire public comme c'était le cas dans les versions précédentes. Il est toujours possible d'inclure des ressources dans les sous répertoires des packages du module mais il faut que celles ci soient spécifiques au module et manipulées dans le code. Pour accéder à une telle ressource, il est maintenant obligatoire d'utiliser la méthode GWT.getModuleBaseURL() pour obtenir le préfixe de l'url de la ressource.

Deux nouveaux outils sont proposés pour exploiter cette nouvelle structure de projet :

- HostedMode : permet l'exécution en mode hosted en remplacement de GWTShell
- Compiler : permet la compilation du code Java en Javascript en remplacement de GWTCompiler

L'application GWTShell utilise un serveur Tomcat embarqué. L'application HostedMode utilise un serveur Jetty embarqué.



La partie graphique cliente possède un nouveau bouton « Restart Server » qui permet de redémarrer le serveur Jetty : cela permet de prendre en compte des modifications dans la partie serveur sans avoir à arrêter et relancer l'application graphique cliente comme c'était le cas avec GWTShell.

65.1.2.2. Un nouveau système de gestion des événements

Le système de gestion des événements par listener est remplacé par un système de gestion par handler.

Les principales différences entre le deux systèmes sont :

- Les méthodes de type EventHandler ne possèdent qu'un seul paramètre de type GwtEvent. Par exemple, la classe ClickHandler possède la méthode onClick(ClickEvent)
- Chaque EventHandler ne possède qu'une seule méthode : ceci évite d'avoir à écrire des méthodes vides mais cela peut nécessiter de remplacer un listener par éventuellement plusieurs handlers selon les besoins

La création de ces propres composants est facilitée car il n'est plus nécessaire de gérer manuellement les listeners. Tous les composants possèdent un objet de type HandlerManager dont le but est de gérer les handlers enregistrés auprès du composant.

La méthode addDomHandler() permet de gérer des événements natifs tel que ClickEvent par exemple : le handler est alors invoqué à l'émission de l'événement.

Exemple :

```
Button bouton = new Button("fermer");

bouton.addClickListener(new ClickListener() {
  @Override
  public void onClick(Widget sender) {
    // traitement du clic sur le bouton

  }
});
```

Exemple :

```
Button bouton = new Button("fermer");
```

```
bouton.addClickHandler(new ClickHandler() {
  @Override
  public void onClick(ClickEvent event) {
    // traitement du clic sur le bouton

  }
});
```

Certains handlers existants ont du être adaptés :

Exemple :

```
final DisclosurePanel panel = new DisclosurePanel("Cliquez pour ouvrir");

panel.addEventHandler(new DisclosureHandler() {
  public void onClose(DisclosureEvent event) {
    panel.getHeaderTextAccessor().setText("Cliquez pour ouvrir");
  }

  public void onOpen(DisclosureEvent event) {
    panel.getHeaderTextAccessor().setText("Cliquez pour fermer");
  }
});

panel.add(new Image("images/logo_java.jpg"));
panel.setWidth("300px");

RootPanel.get("app").add(panel);
```

Exemple :

```
final DisclosurePanel panel = new DisclosurePanel("Cliquez pour ouvrir");

panel.addOpenHandler(new OpenHandler<DisclosurePanel>() {
  @Override
  public void onOpen(OpenEvent<DisclosurePanel> event) {
    panel.getHeaderTextAccessor().setText("Cliquez pour fermer");
  }
});

panel.addCloseHandler(new CloseHandler<DisclosurePanel>() {
  @Override
  public void onClose(CloseEvent<DisclosurePanel> event) {
    panel.getHeaderTextAccessor().setText("Cliquez pour ouvrir");
  }
});

panel.add(new Image("images/logo_java.jpg"));
panel.setWidth("300px");

RootPanel.get("app").add(panel);
```

65.1.2.3. De nouveaux composants

Les composants DatePicker et DateBox permet à l'utilisateur de sélectionner une date dans un calendrier.

Le panneau de type LazyPanel permet de retarder la création des objets pour son rendu lorsqu'il sera affiché pour la première fois ce qui peut permettre d'améliorer le temps de démarrage de l'application qui n'est plus obligé d'instancier les composants pour le rendu de tous les éléments de l'application.

65.1.3. GWT version 1.7

GWT 1.7 est une mise à jour mineure qui apporte un meilleur support pour les dernières versions des navigateurs (Internet Explorer 8, Firefox 3.5 et Safari 4) et corrige quelques bugs majeurs. Elle a été publiée en juillet 2009.

Cette version est téléchargeable à l'url http://code.google.com/p/google-web-toolkit/downloads/list.

Pour un projet en GWT 1.6, il suffit simplement de le recompiler avec la version 1.7, sans modifier le code, pour que l'application soit compatible avec les nouvelles versions des navigateurs supportés.

65.2. La création d'une application

GWT propose plusieurs scripts pour générer des projets GWT composés d'une structure de répertoires et de fichiers fournissant le minimum pour développer un projet.

Pour créer un nouveau projet, il faut créer un nouveau répertoire et utiliser l'application applicationCreator fournie avec GWT qui permet de créer une petite application d'exemple qui peut facilement servir de base pour le développement d'une application utilisant GWT.

La version Windows de GWT contient un script pour lancer l'application ApplicationCreator. Il suffit d'exécuter ce script avec en paramètre le nom pleinement qualifié de la classe principale de l'application. Le dernier package de cette classe doit se nommer obligatoirement client pour éviter une erreur lors de l'exécution de ApplicationCreator

Résultat :

```
D:\gwt-windows-1.3.3>ApplicationCreator com.jmdoudoux.testgwt.monapp
'com.jmdoudoux.testgwt.monapp': Please use 'client' as the final package, as in
'com.example.foo.client.MyApp'.
It isn't technically necessary, but this tool enforces the best practice.
Google Web Toolkit 1.3.3
ApplicationCreator [-eclipse projectName] [-out dir] [-overwrite] [-ignore] clas
sName

where
  -eclipse    Creates a debug launch config for the named eclipse project
  -out        The directory to write output files into (defaults to current)
  -overwrite  Overwrite any existing files
  -ignore     Ignore any existing files; do not overwrite
and
  className   The fully-qualified name of the application class to create
```

Par défaut, les fichiers générés le sont dans le répertoire principal. Pour préciser le répertoire à utiliser, il faut le préciser en utilisant le paramètre -out (celui-ci doit exister)

Résultat :

```
D:\gwt-windows-1.3.3>mkdir MonApp

D:\gwt-windows-1.3.3>ApplicationCreator -out MonApp com.jmdoudoux.testgwt.client
.MonApp
Created directory MonApp\src
Created directory MonApp\src\com\jmdoudoux\testgwt
Created directory MonApp\src\com\jmdoudoux\testgwt\client
Created directory MonApp\src\com\jmdoudoux\testgwt\public
Created file MonApp\src\com\jmdoudoux\testgwt\MonApp.gwt.xml
Created file MonApp\src\com\jmdoudoux\testgwt\public\MonApp.html
Created file MonApp\src\com\jmdoudoux\testgwt\client\MonApp.java
Created file MonApp\MonApp-shell.cmd
Created file MonApp\MonApp-compile.cmd
```

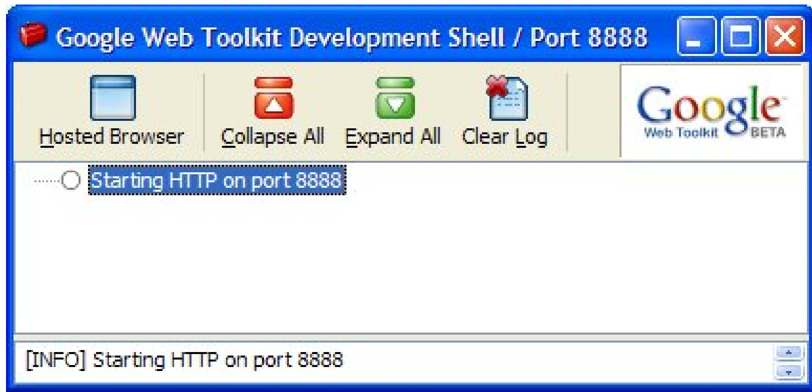
Plusieurs répertoires et fichiers sont créés :

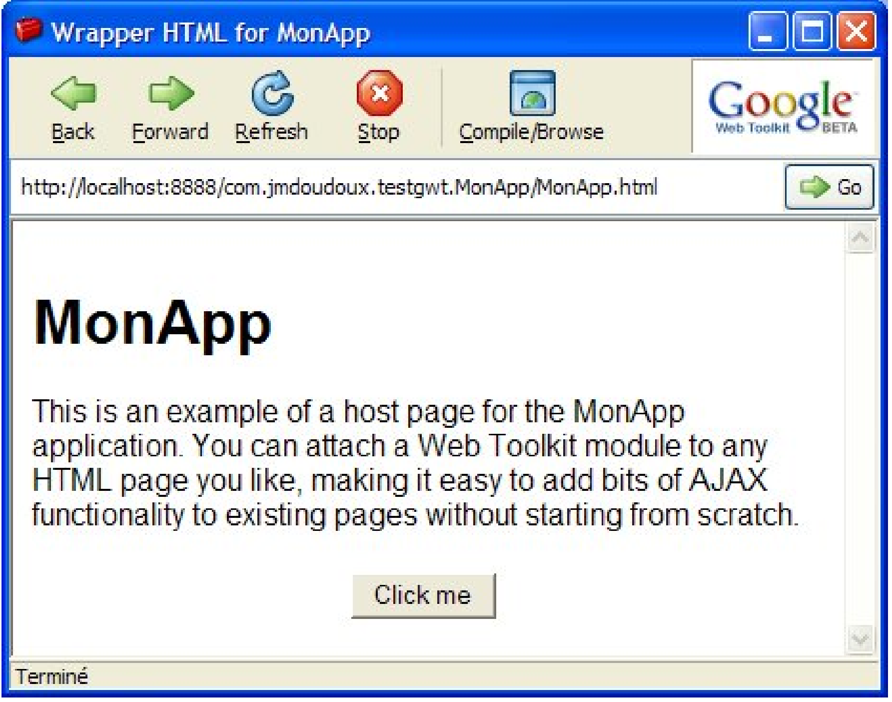
- le répertoire src composé de plusieurs sous répertoires contient les sources de l'application générée : le sous répertoire public contient les pages html et le sous répertoire client contient les sources Java
- le fichier MonApp.gwt.xml contient la configuration de l'application
- le fichier MonApp-shell.cmd permet d'exécuter l'application en mode hôte
- le fichier MonApp-compile.cmd permet de compiler et d'exécuter l'application en mode web

Pour exécuter l'application en mode hôte, il suffit donc de lancer le script MonApp-shell.cmd

Résultat :

```
D:\gwt-windows-1.3.3>cd MonApp

D:\gwt-windows-1.3.3\MonApp>dir
 Le volume dans le lecteur D s'appelle Java
 Le numéro de série du volume est D8C0-0514

 Répertoire de D:\gwt-windows-1.3.3\MonApp

30/07/2007  22:10    <REP>          .
30/07/2007  22:10    <REP>          ..
30/07/2007  22:10               186 MonApp-compile.cmd
30/07/2007  22:10               195 MonApp-shell.cmd
30/07/2007  22:10    <REP>          src
               2 fichier(s)              381 octets
               3 Rép(s)  16 037 978 112 octets libres

D:\gwt-windows-1.3.3\MonApp>MonApp-shell.cmd
```





Pour vérifier la bonne exécution de l'application, il suffit de cliquer sur le bouton "Click me" pour voir apparaître un message.

65.2.1. L'application générée

L'application générée se compose de plusieurs fichiers qui constituent la base de l'application.

Les fichiers sources de l'application sont stockés dans un package qui contient toujours trois sous répertoires :

Package	Rôle
client	Contient les classes qui composent la partie interface graphique de l'application. Seules les classes de ce package et de ses sous packages seront compilées en Java
public	Contient les ressources statiques web : pages HTML, images, Javascript, feuilles de style CSS, ...
server	Contient les classes qui seront exécutées côté serveur

L'application repose sur une page html nommé nom_du_projet.html dans le répertoire public.

Le code de l'application est contenu dans la classe nom_du_projet.java du répertoire client.

Une application GWT est contenue dans un module. La configuration d'un module est le fichier nom_du_projet.gwt.xml.

65.2.1.1. Le fichier MonApp.html

Le fichier MonApp.html contenu dans le sous répertoire public contient la structure de la page de l'application.

Le fichier html d'une application GWT est généralement très simple : la page html sert d'enveloppe pour recevoir les différents composants graphiques qui seront ajoutés grâce à du code Java.

Exemple :

```
<html>
<head>
<title>Wrapper HTML for MonApp</title>
<style>
  body,td,a,div,.p{font-family:arial,sans-serif}
  div,td{color:#000000}
  a:link,.w,.w a:link{color:#0000cc}
  a:visited{color:#551a8b}
  a:active{color:#ff0000}
</style>
<meta name='gwt:module' content='com.jmdoudoux.testgwt.MonApp'>

</head>
<body>
<script language="javascript" src="gwt.js"></script>
<iframe id="__gwt_historyFrame" style="width:0;height:0;border:0"></iframe>
<h1>MonApp</h1>
<p>This is an example of a host page for the MonApp application. You
can attach a Web Toolkit module to any HTML page you like, making it
easy to add bits of AJAX functionality to existing pages without
starting from scratch.</p>
<table align=center>
  <tr>
    <td id="slot1"></td>
    <td id="slot2"></td>
  </tr>
</table>
</body>
</html>
```

Les deux balises <td> possèdent un identifiant distinct : ils définissent des conteneurs dans lesquels les composants vont être ajoutés. L'identifiant sera utilisé dans le code Java pour obtenir une référence sur le conteneur.

Le script Javascript gwt.js est utilisé pour lancer l'application notamment en exécutant la version de l'application dédiée au navigateur utilisé.

L'iframe est utilisé dans le mécanisme de gestion de l'historique de navigation.

65.2.1.2. Le fichier MonApp.gwt.xml

Ce fichier contient la définition et la configuration du module notamment :

- la classe qui fait office de point d'entrée dans l'application
- les dépendances
- les directives de compilation

C'est un fichier xml dont l'extension est .gwt.xml. Le tag racine est le tag <module>

Le tag <inherits> permet de préciser les fonctionnalités de base qui composeront le module.

Le tag <entry-point> permet de préciser la classe pleinement qualifiée qui est le point d'entrée de l'application : l'attribut class permet de préciser le nom de la classe principale de l'application.

Exemple :

```
<module>

  <!-- Inherit the core Web Toolkit stuff.                  -->
  <inherits name='com.google.gwt.user.User' />

  <!-- Specify the app entry point class.                   -->
  <entry-point class='com.jmdoudoux.testgwt.client.MonApp' />

</module>
```

65.2.1.3. Le fichier MonApp.java

La classe MonApp contient le code de l'application avec notamment :

- la définition des composants de l'interface graphique
- la définition des gestionnaires d'événements (listeners ou handlers) pour répondre aux actions de l'utilisateur
- les traitements en réponse aux événements

La méthode onModuleLoad() est le point d'entrée de l'application. Cette méthode contient la définition de l'IHM de l'application.

La mise en oeuvre des gestionnaires d'événements est similaire à celle de la mise en oeuvre d'autres framework permettant le développement d'interfaces graphiques tels que AWT, Swing ou SWT. Elle repose sur l'enregistrement de listeners généralement définis sous la forme de classes anonymes.

Exemple :

```
package com.jmdoudoux.testgwt.client;

import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.user.client.ui.Button;
import com.google.gwt.user.client.ui.ClickListener;
```

```
import com.google.gwt.user.client.ui.Label;
import com.google.gwt.user.client.ui.RootPanel;
import com.google.gwt.user.client.ui.Widget;

/**
 * Entry point classes define <code>onModuleLoad()</code>.
 */
public class MonApp implements EntryPoint {

  /**
   * This is the entry point method.
   */
  public void onModuleLoad() {
    final Button button = new Button("Click me");
    final Label label = new Label();

    button.addClickListener(new ClickListener() {
      public void onClick(Widget sender) {
        if (label.getText().equals(""))
          label.setText("Hello World!");
        else
          label.setText("");
      }
    });

    // Assume that the host HTML has elements defined whose
    // IDs are "slot1", "slot2".  In a real app, you probably would not want
    // to hard-code IDs.  Instead, you could, for example, search for all
    // elements with a particular CSS class and replace them with widgets.
    //
    RootPanel.get("slot1").add(button);
    RootPanel.get("slot2").add(label);
  }
}
```

Important : les classes de la partie client ne peuvent pas faire référence aux classes de la partie serveur.

65.3. Les modes d'exécution

Une application GWT peut être exécutée dans deux modes :

- Le mode hôte (hosted mode) : ce mode est utilisé pour le développement et la mise au point de l'application car il permet la mise en oeuvre d'un débogueur
- Le mode web (web mode) : ce mode est utilisé pour le déploiement et l'exploitation de l'application par les utilisateurs

65.3.1. Le mode hote (hosted mode)

Dans ce mode, l'application est exécutée de façon hybride sous la forme de code Java exécuté dans un navigateur spécial. Ceci permet notamment l'utilisation du débogueur d'un IDE.

L'environnement d'exécution est composé d'une console, d'un conteneur web (Tomcat ou Jetty selon la version de GWT) et d'un navigateur dédié (Internet Exploreur sous Windows et Firefox sous Linux).

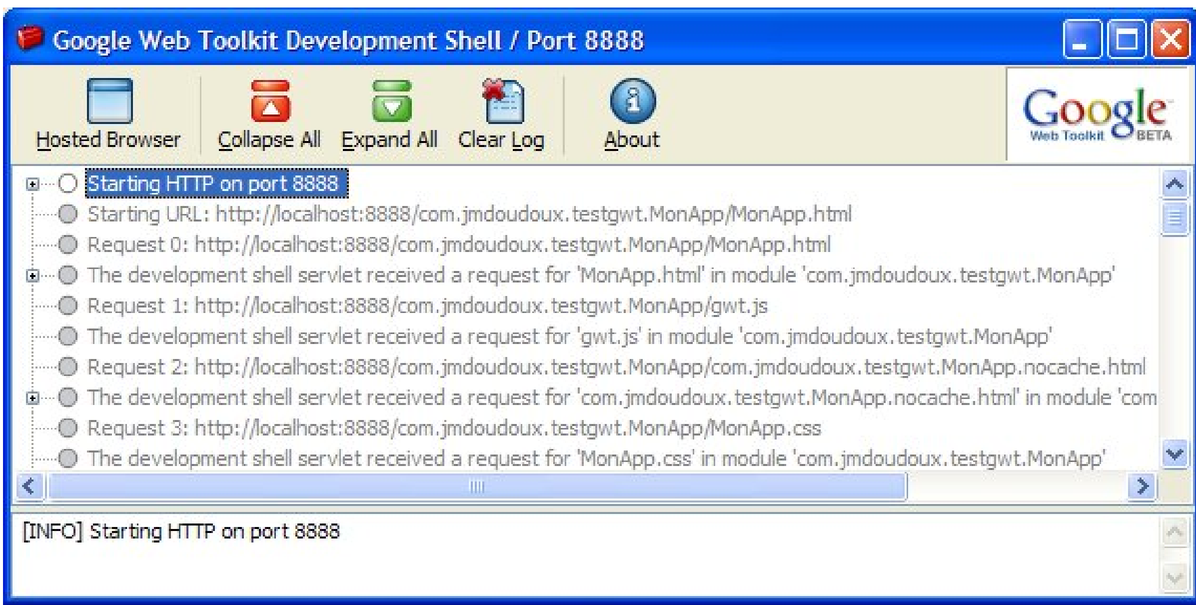
Remarque : le mode hote n'est disponible que sous Windows et Linux.

Pour exécuter une application dans le mode hôte, il faut exécuter le script dont le nom se compose du nom de l'application et se termine par -shell. Plusieurs options peuvent être fournies à l'environnement d'exécution :

- -noserver
- -out

* -gen
* -logLevel level : permet de préciser niveau de trace. Level peut prendre les valeurs : ERROR, WARN, INFO, TRACE, DEBUG, SPAM, ALL

Exemple avec l'option -logLevel ALL



Le mode hôte, facilite grandement l'écriture et la mise au point de l'application en permettant :

* l'exécution de l'application
* la modification du code source, sa recompilation
* de relancer l'application simplement en cliquant sur le bouton Refresh

L'environnement d'exécution affiche deux fenêtres :

* la fenêtre "Google Web Toolkit Development Shell / Port 8888" : affiche les messages du serveur et permet d'interagir avec lui
* le navigateur qui affiche l'application

Lorsque l'application est exécutée en mode hote :

* il n'y a pas besoin de compiler et déployer l'application à chaque modification
* pour tester une modification faite dans le code et compilée en byte code, il suffit simplement de cliquer sur le bouton de rafraichissement du navigateur : ceci permet de tester rapidement des modifications
* il est possible d'utiliser le débogueur d'un IDE en positionnant des points d'arrêts

65.3.2. Le mode web (web mode)

Dans le mode web, la partie cliente de l'application doit être compilée en Javascript. Un script de compilation est généré lors de la création de l'application. Le nom de ce script est composé du nom de l'application suivi de « -compile.cmd ».

Le compilateur possède plusieurs options :

* -logLevel
* -treeLogger
* -gen
* -out
* -style : style de lisibilité du code généré : OBFUSCATED (style par défaut), PRETTY ou DETAILED

Le compilateur génère plusieurs fichiers correspondant à chaque navigateur supporté par le compilateur et éventuellement un pour chaque langue mise en oeuvre pour internationaliser l'application. Ceci permet de réduire la taille

du fichier Javascript de l'application car elle ne contient que du code pour le navigateur et la locale utilisée.

Résultat :

```
D:\gwt-windows-1.3.3\MonAppProjet>MonApp-compile.cmd
Output will be written into D:\gwt-windows-1.3.3\MonAppProjet\www\com.jmdoudoux.
testgwt.MonApp
Copying all files found on public path
Compilation succeeded
```

Le répertoire www est créé : il contient un sous répertoire qui porte le nom du package principal de l'application. Ce répertoire contient les fichiers générés.

Résultat : le contenu du répertoire de D:\gwt-windows-1.3.3\MonAppProjet\www\com.jmdoudoux.testgwt.MonApp

```
D:\gwt-windows-1.3.3\MonAppProjet\www\com.jmdoudoux.testgwt.MonApp>dir
 Le volume dans le lecteur D s'appelle Java
 Le numéro de série du volume est D8C0-0514
13/08/2007  23:03    <REP>          .
13/08/2007  23:03    <REP>          ..
13/08/2007  23:03            38 805 0A3A82524C61CDBB6FEA7286E3F85391.cache.html
13/08/2007  23:03               801 0A3A82524C61CDBB6FEA7286E3F85391.cache.xml
13/08/2007  23:03            38 865 4D7E1609F2BC0A4229FFC55F98976B68.cache.html
13/08/2007  23:03               798 4D7E1609F2BC0A4229FFC55F98976B68.cache.xml
13/08/2007  23:03            38 628 921FF51D706A4D67E10268B5DD22D7D7.cache.html
13/08/2007  23:03               801 921FF51D706A4D67E10268B5DD22D7D7.cache.xml
13/08/2007  23:03            38 422 942F11931D582C6DF0166C3EA388BE17.cache.html
13/08/2007  23:03               798 942F11931D582C6DF0166C3EA388BE17.cache.xml
13/08/2007  23:03             2 900 com.jmdoudoux.testgwt.MonApp.nocache.html
13/08/2007  23:03            17 336 gwt.js
13/08/2007  23:03               444 history.html
13/08/2007  23:03               501 MonApp.css
13/08/2007  23:03               512 MonApp.html
13/08/2007  23:03                82 tree_closed.gif
13/08/2007  23:03                78 tree_open.gif
13/08/2007  23:03                61 tree_white.gif
              16 fichier(s)          179 832 octets
```

Les fichiers .cache.html contiennent le code Javascript pour chaque navigateur. Le nom du fichier est encrypté. Chacun de ces fichiers possède un fichier avec le même nom et l'extension .cache.xml.

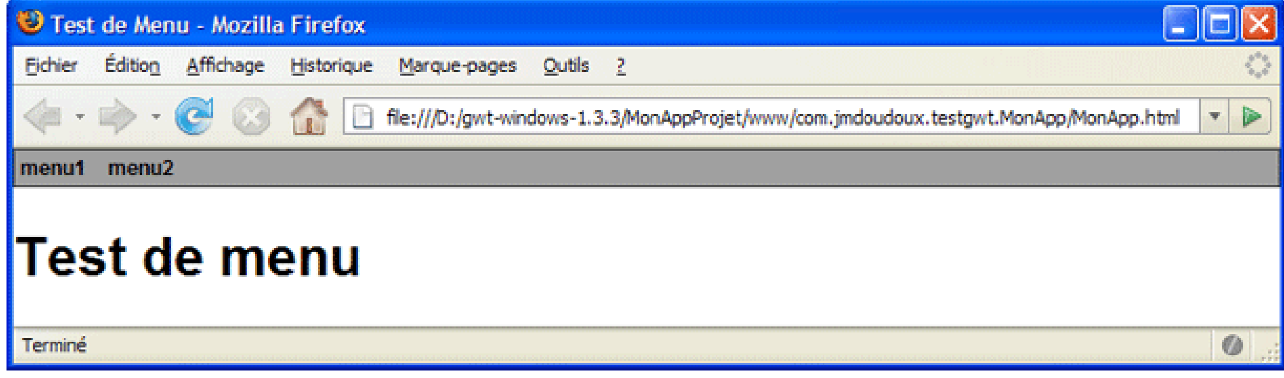
Ces fichiers donnent des informations sur le contenu des fichiers

Exemple : le fichier4D7E1609F2BC0A4229FFC55F98976B68.cache.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<cache-entry>
        <rebind-decision in="com.google.gwt.user.client.ui.impl.TextBoxImpl"
          out="com.google.gwt.user.client.ui.impl.TextBoxImpl"/>
        <rebind-decision in="com.google.gwt.user.client.impl.DOMImpl"
          out="com.google.gwt.user.client.impl.DOMImplMozilla"/>
        <rebind-decision in="com.google.gwt.user.client.impl.HistoryImpl"
          out="com.google.gwt.user.client.impl.HistoryImplStandard"/>
        <rebind-decision in="com.google.gwt.user.client.ui.impl.FormPanelImpl"
          out="com.google.gwt.user.client.ui.impl.FormPanelImpl"/>
        <rebind-decision in="com.jmdoudoux.testgwt.client.MonApp"
          out="com.jmdoudoux.testgwt.client.MonApp"/>
        <rebind-decision in="com.google.gwt.user.client.ui.impl.PopupImpl"
          out="com.google.gwt.user.client.ui.impl.PopupImpl"/>
</cache-entry>
```

Ainsi, le fichier 4D7E1609F2BC0A4229FFC55F98976B68 correspond au code pour le navigateur Mozilla.

L'ouverture du fichier MonApp.html dans un navigateur lance l'application



Pour diffuser l'application qui ne contient pas de partie serveur, il suffit de copier les fichiers générés dans un serveur web hormis les fichiers .xml.

Pour une application qui contient une partie serveur, il faut packager l'application dans un war. Cette archive doit contenir :

- la partie cliente : les fichiers HTML et Javascript générés ainsi que les ressources statiques (CSS, images, ...)
- la partie serveur : les fichiers .class, le fichier web.xml, les bibliothèques requises (gwt-servlet.jar, ...)

65.4. Les éléments de GWT

GWT se compose de plusieurs éléments :

- le compilateur qui compile du code Java en code Javascript
- JSNI qui permet l'utilisation de code Javascript dans le code Java
- JRE Emulation Library qui est un sous ensemble des classes de base de Java
- une API qui fournit de nombreuses fonctionnalités : composants graphiques pour IHM, appels RPC vers un serveur, gestion de l'historique de navigation, parser de document XML, tests unitaires avec JUnit, ...

65.4.1. Le compilateur

Le compilateur GWT de code Java en Javascript est encapsulé dans la classe com.google.gwt.dev.GWTCompiler.

Le compilateur traite l'entry point pour chacune de ces dépendances. Le compilateur utilise les fichiers sources mais n'utilise pas les fichiers .class.

Le compilateur des versions antérieures à la version 1.5 de GWT ne supporte que du code source respectant la syntaxe de Java 1.4 : les fonctionnalités de Java 5 ne sont donc pas supportées avant la version 1.5 de GWT. Cette restriction n'est valable que pour le code de la partie cliente qui sera transformé en Javascript. La partie serveur n'est pas concernée par cette restriction puisqu'elle sera compilée en byte code pour être exécutée dans la JVM du serveur.

Le compilateur génère un fichier Javascript par navigateur et par langage si l'internationalisation est utilisée dans l'application. Le fichier pour le navigateur concerné sera chargé au lancement de l'application. L'intérêt majeur est de limiter le code Javascript uniquement à celui pour le navigateur utilisé : ceci évite d'avoir à gérer de nombreuses opérations de tests sur le navigateur comme cela est fréquent dans le code Javascript.

Cette fonctionnalité est aussi mise en oeuvre pour chaque langue utilisée pour internationaliser l'application. Le code Javascript ne contient que les libellés pour la langue du fichier.

Le compilateur peut mettre en oeuvre des techniques d'obfuscation du code Javascript généré afin de le protéger et surtout de réduire sa taille.

Au final, le code contenu dans chaque fichier généré est le plus réduit possible.

65.4.2. JRE Emulation Library

Pour utiliser certaines classes de la bibliothèque de base de Java, GWT propose le JRE Emulation Library qui contient certaines classes fréquemment utilisées dans les applications. Le JRE Emulation Library de GWT permet d'utiliser certaines des classes de base de la bibliothèque de Java. Ces classes sont un sous ensemble de la bibliothèque correspondant à celles qui peuvent être transformées en Javascript par le compilateur.

Les classes du package java.lang incluses dans le JRE Emulation Library sont :

Boolean	Byte	Character
Class	Double	Float
Integer	Long	Math
Number	Object	Short
String	StringBuffer	System
Throwable	Error	Exception

Il existe aussi des différences entre leur utilisation dans une JVM et dans le JRE Emulation Library. Ces différences sont essentiellement imposées par la conversion du code en Javascript.

Certaines fonctionnalités de ces classes sont ainsi différentes de leurs homologues de la bibliothèque Java, par exemple :

- System.out et System.err sont utilisables dans le mode hosted mais n'ont aucun effet dans le mode web
- les expressions régulières utilisées dans certaines méthodes la classe String (replaceAll(), replaceFirst()) sont différentes
- ...

Attention : Javascript ne propose pas de support pour les entiers sur 64 bits représentés par une variable de type long en Java. Le compilateur transforme les types long en double. Le fonctionnement de l'application peut donc être différent dans le mode host et web.

La méthode getStackTrace() de la classe Throwable n'est pas utilisable dans le mode web.

Les assertions (mot clé assert) sont ignorées par le compilateur.

Javascript n'est pas multithread : tout ce qui concerne le multithreading dans le langage Java est donc inutilisable et ignoré par le compilateur.

L'API reflexion permettant une utilisation dynamique des objets n'est pas utilisable. L'API GWT propose uniquement la méthode GWT.getTypeName() : elle renvoie une chaîne de caractères qui correspond au type de l'objet fourni en paramètre.

Javascript ne propose pas le support pour la finalisation des objets lors de leur traitement par le garbage collector.

Javascript ne propose pas le support d'une précision constante dans les calculs en virgule flottante. Il n'est donc pas recommandé d'effectuer de tel calcul dans la partie cliente. Des calculs en virgule flottante peuvent être réalisés mais leur précision n'est pas garantie.

La sérialisation proposée par Java n'est pas supportée par GWT qui propose son propre mécanisme pour les appels de type RPC vers le serveur.

Pour des raisons de performance, il n'est pas recommandé d'utiliser des objets de type Long, Float ou Double comme clé pour des objets de type Map.

Les classes du package java.lang incluses dans le JRE Emulation Library sont :

AbstractCollection	AbstractList	AbstractMap
AbstractSet	ArrayList	Arrays
Collections	Date	HashMap
Stack	Vector	Collection
Comparator	EventListener	Iterator
List	Map	RandomAccess
Set		

65.4.3. Les modules

La classe qui est le point d'entrée d'un module doit implémenter l'interface EntryPoint. Cette interface définit la méthode void onLoadModule().

Un module contient un fichier descripteur au format XML. Son nom est composé du nom du module suivi de .gwt.xml

Il permet de préciser :

- Les autres modules utilisés
- Le nom de la classe qui sert de point d'entrée
- Les chemins des fichiers source qui doivent être compilés en Javascript
- Les chemins pour trouver les ressources publiques (CSS, images, javasript, ...)

Ce fichier est stocké dans le répertoire qui contient la classe qui sert de point d'entrée au module.

Le tag <module> est le tag racine.

Le tag <inherits> permet de préciser un autre module qui sera utilisé. L'attribut name permet de préciser le nom du module.

Le tag <source> permet de préciser le répertoire qui contient des sources à compiler. Le répertoire est précisé grâce à l'attribut path.

Le tag <stylesheet> permet de préciser une feuille de style CSS. L'attribut src permet de préciser le nom du fichier CSS.

Le tag <servlet> permet de définir une servlet qui sera utilisée pour les communications de type RPC avec le serveur en mode hosted. L'attribut path permet de préciser l'uri de la servlet. L'attribut class permet de préciser le nom pleinement qualifié de la classe qui encapsule la servlet.

65.4.4. Les limitations

Le code de l'application est compilée en Javascript : seules les fonctionnalités compilables en Javascript peuvent être utilisées. Ainsi par exemple, il n'est pas possible d'utiliser le type primitif long puisque Javascript ne supporte pas le 64 bits. Cependant le code se compile parfaitement puisque chaque variable de type long est convertie en type double ce qui peut provoquer des effets de bord.

Javascript n'est pas multi thread : il faut en tenir compte lors du développement de l'application.

65.5. L'interface graphique des applications GWT

Pour la partie graphique de l'application, GWT propose un ensemble de composants de deux types :

- widgets : ce sont des contrôles utilisateurs soit de base (boutons, zone de texte, case à cocher, bouton radio, ...) soit plus riche en fonctionnalités (barre de menu, onglet, treeview, ...)
- panels : ces composants se chargent d'assurer la disposition des composants qui leurs sont rattachés à l'image des layouts manager de Swing. Certains panels proposent aussi de l'interactivité avec l'utilisateur.

En plus de ces composants proposés en standard par GWT, il est possible de développer ces propres composants.

Il existe plusieurs projets open source qui développent d'autres composants (calendrier, grille, ...) ou des composants qui encapsulent des bibliothèques Javascript existantes (Scriptaculous, Google Search et Map, ...).

Les composants possèdent une double représentation :

- en Java, lors de l'écriture du code et de l'exécution de l'application dans le mode hôte
- dans l'arbre DOM de la page une fois le code compilé en Javascript et exécuté dans le mode web

L'organisation des composants n'est pas assurée par des layouts mais par des panneaux qui sont plus facilement adaptables à leur rendu en HTML. Par exemple :

- HorizontalPanel : les composants sont mis les uns à coté des autres de gauche à droite
- FlowPanel : arrange les composants qu'il contient les uns à côté des autres en allant du haut à gauche vers le bas à droite
- AbsolutePanel : permet de préciser les coordonnées des composants
- ...

GWT propose un ensemble complet de composants graphiques (widgets) et panneaux (panels).

L'état de l'interface graphique est maintenu sur le client dans une application GWT.

Exemple :

```
package com.jmdoudoux.test.gwt.client;

import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.user.client.ui.Button;
import com.google.gwt.user.client.ui.ClickListener;
import com.google.gwt.user.client.ui.FlowPanel;
import com.google.gwt.user.client.ui.Panel;
import com.google.gwt.user.client.ui.RootPanel;
import com.google.gwt.user.client.ui.TextBox;
import com.google.gwt.user.client.ui.Widget;

public class MainEntryPoint implements EntryPoint {

  private boolean etat = true;

  public MainEntryPoint() {
  }

  public void onModuleLoad() {
    final TextBox text = new TextBox();
    text.setText("AAAAA");
    final Button button = new Button();
    button.setText("Inverser");
    button.addClickListener(new ClickListener() {

      public void onClick(Widget sender) {
        etat = !etat;
        if (etat) {
          text.setText("AAAAA");
        } else {
          text.setText("ZZZZZ");
        }
      }
```

```
    });
    Panel main = new FlowPanel();
    RootPanel.get().add(main);
    main.add(text);
    main.add(button);
  }
}
```

AAAAA Inverser

Un clic le bouton "Inverser" inverse les lettres affichées

ZZZZZ Inverser

65.6. La personnalisation de l'interface

Comme une application GWT est compilée pour générer une application utilisant le DHTML et Javascript, la personnalisation de l'interface de l'application repose sur les feuilles de style CSS.

Le rendu des composants d'une application GWT peut donc être assuré via des styles CSS.

La plupart des composants ayant un rendu graphique possèdent une classe de style CSS par défaut composée de gwt- suivi du nom du composant (exemple : gwt-Button, gwt-CheckBox, ...).

Il est possible d'utiliser une feuille de style CSS définie dans un fichier stocké dans le sous répertoire public de l'application.

Exemple : le fichier monstyle.css

Résultat :

```
root {
    display: block;
}

.message
{
    color: blue;
    display: block;
    width: 450px;
    padding: 2px 4px;
    margin-top: 3px;
    text-decoration: none;
    text-align: center;
    font-family: Verdana,Arial,Helvetica,sans-serif;
    font-size: 10px;
    border: 1px solid;
    border-color: black;
    ext-decoration: none;
}

.erreur
{
    color: white;
    display: block;
    width: 450px;
    padding: 2px 4px;
    margin-top: 3px;
    text-decoration: none;
    text-align: center;
    font-family: Verdana,Arial,Helvetica,sans-serif;
    font-size: 10px;
```

```
    font-weight: bold;
    border: 1px solid;
    border-color: black;
    ext-decoration: none;
    background-color: red;
}
```

Pour que la feuille de style CSS soit prise en compte par l'application, il faut la déclarer dans le fichier de configuration du module. Cette déclaration se fait à l'aide du tag <stylesheet>. Son attribut src permet de préciser le nom du fichier CSS.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<module>
  <inherits name="com.google.gwt.user.User"/>
  <entry-point class="com.jmdoudoux.test.gwt.client.MainEntryPoint"/>
  <stylesheet src="monstyle.css"/>
</module>
```

Chaque composant possède plusieurs méthodes relatives aux styles CSS héritées de la classe UIObject :

- addStyleName() : permet d'ajouter un style à la liste des styles du composant
- setStylePrimaryName() :
- setStyleName() : permet de forcer le sélecteur de classe du style utilisé en supprimant tous les styles appliqués

Exemple :

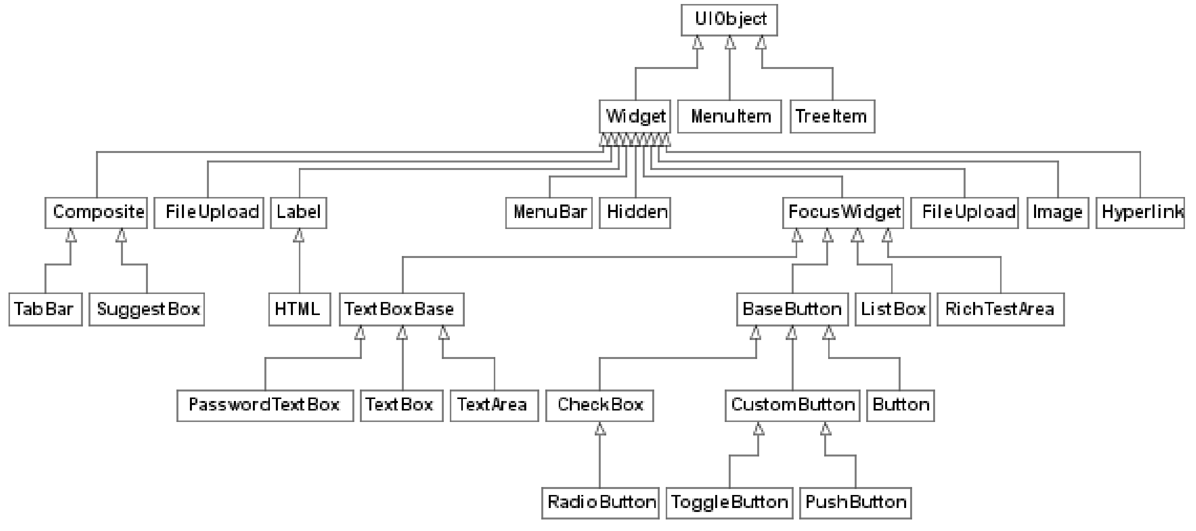
```
...
            public void onSuccess(Object result) {
                lblMessage.setStyleName("message");
                lblMessage.setText((String) result);
            }

            public void onFailure(Throwable caught) {
                lblMessage.setStyleName("erreur");
                lblMessage.setText("Echec de la communication");
            }
...
```

65.7. Les composants (widgets)

GWT propose un ensemble complet de composants graphiques de base pour le développement de l'interface graphique d'une application.

Tous les composants héritent de la classe com.google.gwt.user.client.ui.UIObject.

La classe UIObject est la super classe des classes Widget, MenuItem et TreeItem. La classe Widget est la super classe de la quasi totalité des composants graphiques de GWT.



Composant	Description	Version de GWT	Classe CSS par défaut
Button	Bouton	1.0	gwt-Button
ButtonBase	Classe mère des boutons		
CheckBox	Case à cocher	1.0	gwt-CheckBox
Composite	Classe qui permet de créer un nouveau composant par assemblage		
DateBox	Zone de saisie de texte qui ouvre un DatePicker	1.6	
DatePicker	Permet de sélectionner une date	1.6	
FileUpload	Elément HTML de type <input type="file">	1.1	
FocusWidget	Classe mère des composants pouvant avoir le focus		
Hidden	Champ de type HIDDEN dans un formulaire HTTP	1.2	
HTML	Contient du code HTML	1.0	gwt-HTML
Hyperlink	Hyper lien	1.1	gwt-Hyperlink
Image	Image	1.1	gwt-Image
Label	Zone de texte		gwt-Label
ListBox	Liste ou liste déroulante		gwt-ListBox
MenuBar	Bar de menu	1.0	gwt-MenuBar
MenuItem	Elément d'une barre de menu	1.0	gwt-MenuItem
PasswordTextBox	Zone de saisie de texte masqué		gwt-PasswordTextBox
RadioButton	Bouton radio		gwt-RadioButton
RichTextArea	Zone de saisie de texte riche	1.4	
SuggestBox	Zone de saisie de texte qui suggère des valeurs selon la saisie	1.4	
TabBar	Une barre d'onglets	1.0	gwt-TabBar gwt-TabBarFirst gwt-TabBarRest gwt-TabBarItem gwt-TabBarItem-selected

TextArea	Zone de saisie de texte multi-ligne	1.0	gwt-TextArea
TextBox	Zone de saisie de texte mono-ligne	1.0	gwt-TextBox
TextBoxBase	Classe mère des zones de saisie de texte		
ToogleButton			
Tree	Treeview	1.0	gwt-Tree
TreeItem	Elément d'un composant treeview	1.0	gwt-TreeItem, gwt-TreeItem-selected
UIObject	Classe mère des éléments graphiques		
Widget	Classe mère des composants		

65.7.1. Les composants pour afficher des éléments

GWT propose plusieurs composants graphiques pour afficher du texte ou des images.

65.7.1.1. Le composant Image

La classe Image encapsule une image qui sera affichée. Dans l'arbre Dom, ce composant est un tag <IMG> d'HTML.

Exemple :

```
    Image image = new Image("images/logo_java.jpg");

    RootPanel.get("app").add(image);
```

Il est possible de gérer plusieurs événements émis par ce composant.

La classe Image possède plusieurs listeners utilisables jusqu'à GWT 1.5 : ClickListener, LoadHandler, MouseListener et MouseWheelListener .

Exemple :

```
    Image image = new Image("images/logo_java.jpg");
    final int largeur = image.getWidth();
    final int hauteur = image.getHeight();
    image.setSize(""+(largeur/2), ""+(hauteur/2));
    image.addClickListener(new ClickListener() {
      @Override
      public void onClick(Widget sender) {
        Image img = (Image) sender;
        if (img.getWidth() == largeur) {
          img.setSize(""+(largeur/2), ""+(hauteur/2));
        } else {
          img.setSize(""+largeur, ""+hauteur);
        }
      }
    } );

    RootPanel.get("app").add(image);
```

A partir de GWT 1.6, le composant Image propose plusieurs handlers : DomHandler, ClickHandler, ErrorHandler, LoadHandler, MouseDownHandler, MouseMoveHandler, MouseOutHandler, MouseOverHandler, MouseUpHandler et MouseWheelHandler.

Exemple :

```
Image image = new Image("images/logo_java.jpg");
final int largeur = image.getWidth();
final int hauteur = image.getHeight();
image.setSize(""+(largeur/2), ""+(hauteur/2));
image.addClickHandler(new ClickHandler() {
  @Override
  public void onClick(ClickEvent event) {
    Image img = (Image) event.getSource();
    if (img.getWidth() == largeur) {
      img.setSize(""+(largeur/2), ""+(hauteur/2));
    } else {
      img.setSize(""+largeur, ""+hauteur);
    }
  }
});

RootPanel.get("app").add(image);
```

65.7.1.2. Le composant Label

La classe Label encapsule un simple texte qui est affiché.

Exemple :

```
Label label = new Label("un libelle");

RootPanel.get("app").add(label);
```

Il est possible de gérer plusieurs événements émis par ce composant.

La classe Label possède trois listeners utilisables jusqu'à GWT 1.5 : ClickListener, MouseListener et MouseWheelListener .

Exemple :

```
Label label = new Label("un libelle");
ClickListener listener = new ClickListener() {
  @Override
  public void onClick(Widget sender) {
    Window.alert("clic sur le libelle");
  }
};

label.addClickListener(listener);

RootPanel.get("app").add(label);
```

A partir de GWT 1.6, le composant Label propose plusieurs handlers : ClickHandler, MouseDownHandler, MouseMoveHandler, MouseOutHandler, MouseOverHandler, MouseUpHandler et MouseWheelHandler.

Exemple :

```
final Label label = new Label("un libelle");
MouseOverHandler mouseOverHandler = new MouseOverHandler() {
  @Override
  public void onMouseOver(MouseOverEvent event) {
    label.setStyleName("label-over");
  }
};

MouseOutHandler mouseOutHandler = new MouseOutHandler() {
```

```
      @Override
      public void onMouseOut(MouseOutEvent event) {
        label.removeStyleName("label-over");
      }
    };

    label.addMouseOverHandler(mouseOverHandler);
    label.addMouseOutHandler(mouseOutHandler);

    RootPanel.get("app").add(label);
```

65.7.1.3. Le composant DialogBox

La classe DialogBox encapsule une fenêtre de type boîte de dialogue.

Exemple :

```
    private static class MessageInfoBox extends DialogBox {

    private MessageInfoBox(String message) {
      setText("Information");
      final DockPanel panel = new DockPanel();
      panel.setVerticalAlignment(HasAlignment.ALIGN_MIDDLE);
      panel.setHorizontalAlignment(HasAlignment.ALIGN_LEFT);
      panel.setStyleName("alignement-gauche");
      panel.add(new Label(message), DockPanel.CENTER);
      panel.add(new Image("imgages/information.jpg"), DockPanel.WEST);

      SimplePanel panelBouton = new SimplePanel();
      panelBouton.setStyleName("alignement-droite");
      Button boutonOk = new Button("OK");
      boutonOk.setWidth("120px");
      boutonOk.addClickHandler(new ClickHandler() {
        public void onClick(ClickEvent event) {
          MessageInfoBox.this.hide();
        }
      });
      panelBouton.add(boutonOk);
      panel.add(panelBouton, DockPanel.SOUTH);
      setWidget(panel);
    }

    public static void afficher(String message) {
      new MessageInfoBox(message).center();
    }
  }

  public void onModuleLoad() {
    ClickHandler handler = new ClickHandler() {
      @Override
      public void onClick(ClickEvent event) {
        MessageInfoBox.afficher("Ceci est une boite de dialogue d'information");
        }
    };
    Button bouton = new Button("Afficher");
    bouton.addClickHandler(handler);
    RootPanel.get("app").add(bouton);
  }
```

L'instance du panneau n'a pas besoin d'être rattachée au RootPanel ou à tout autre composant.

Les méthodes center() ou show() permettent d'afficher le panneau et la méthode hide() permet de le masquer.

Contrairement à un PopupPanel, il est possible de préciser la taille d'un DialogBox sans avoir besoin d'ajouter de composants en utilisant les méthodes setWitdh() et setHeight().

Avant GWT version 1.6, la gestion des événements se fait avec des listeners. Un seul listener est défini pour ce composant : PopupListener

A partir GWT version 1.6, la gestion des événements se fait avec des handlers. Plusieurs handlers sont utilisables avec ce composant : DomHandler et CloseHandler.

65.7.2. Les composants cliquables

65.7.2.1. La classe Button

La classe Button encapsule un bouton : elle hérite de la classe ButtonBase.

Le style CSS associé est .gwt-Button

Avant GWT version 1.6, la gestion des événements se fait avec des listeners. Plusieurs listeners sont utilisables avec ce composant : ClickListener, FocusListener et KeyboardListener.

Exemple :

```
    ClickHandler handler = new ClickHandler() {
      @Override
      public void onClick(ClickEvent event) {
        Window.alert("Bonjour");
      }
    };
    Button bouton= new Button("Afficher");
    bouton.addClickHandler(handler);
    RootPanel.get("app").add(bouton);
```

A partir GWT version 1.6, la gestion des événements se fait avec des handlers. Plusieurs handlers sont utilisables avec ce composant : BlurHandler, ClickHandler, DomHandler, FocusHandler, KeyDownHandler, KeyPressHandler, KeyUpHandler, MouseDownHandler, MouseMoveHandler, MouseOutHandler, MouseOverHandler, MouseUpHandler et MouseWheelHandler.

Exemple :

```
    ClickListener listener = new ClickListener() {
         @Override
         public void onClick(Widget sender) {
                Window.alert("Bonjour");
        }
    };
    Button bouton = new Button("Afficher", listener);
    RootPanel.get("app").add(bouton);
```

65.7.2.2. La classe PushButton

La suite de cette section sera développée dans une version future de ce document

65.7.2.3. La classe ToggleButton

La suite de cette section sera développée dans une version future de ce document

65.7.2.4. La classe CheckBox

La classe CheckBox encapsule un bouton de type case à cocher : elle hérite de la classe ButtonBase.

Le style CSS associé est .gwt-CheckBox et .gwt-CheckBox-disabled

Avant GWT version 1.6, la gestion des événements se fait avec des listeners. Plusieurs listeners sont utilisables avec ce composant : ClickListener, FocusListener, KeyboardListener, MouseListener et MouseWheelListener.

Exemple :

```
ClickListener listener = new ClickListener() {
   @Override
   public void onClick(Widget sender) {
    CheckBox cb = (CheckBox) sender;
    if (cb.isChecked()) {
      Window.alert("Bonjour");
    }
  }
};

CheckBox bouton = new CheckBox("Afficher");
bouton.setChecked(false);
bouton.addClickListener(listener);
RootPanel.get("app").add(bouton);
```

A partir GWT version 1.6, la gestion des événements se fait avec des handlers.Plusieurs handlers sont utilisables avec ce composant : BlurHandler, ClickHandler, DomHandler, FocusHandler, KeyDownHandler, KeyPressHandler, KeyUpHandler, MouseDownHandler, MouseMoveHandler, MouseOutHandler, MouseOverHandler, MouseUpHandler, et MouseWheelHandler

Exemple :

```
ClickHandler handler = new ClickHandler() {
     @Override
     public void onClick(ClickEvent event) {
    CheckBox cb = (CheckBox) event.getSource();
    if (cb.getValue()) {
      Window.alert("Bonjour");
    }
  }
};

CheckBox bouton = new CheckBox("Afficher");
bouton.setValue(false);
bouton.addClickHandler(handler);
RootPanel.get("app").add(bouton);
```

65.7.2.5. La classe RadioButton

La classe RadioButton encapsule un bouton radio : un seul bouton peut être sélectionné parmi ceux d'un même groupe.

Elle hérite de la classe CheckButton.

Le style CSS associé est .gwt-RadioButton

Pour déterminer ou modifier l'état du bouton, il faut utiliser la propriété value (la propriété checked est deprecated).

Exemple :

```
RadioButton rb1 = new RadioButton("valeurs", "Valeur 1");
RadioButton rb2 = new RadioButton("valeurs", "Valeur 2");
RadioButton rb3 = new RadioButton("valeurs", "Valeur 3");

rb2.setValue(true);
VerticalPanel panel = new VerticalPanel();
panel.add(rb1);
panel.add(rb2);
panel.add(rb3);

RootPanel.get("app").add(panel);
```

Avant GWT version 1.6, la gestion des événements se fait avec des listeners. Plusieurs listeners sont utilisables avec ce composant : ClickListener, FocusListener, KeyboardListener, MouseListener et MouseWheelListener.

Exemple :

```
RadioButton radioButton1 = new RadioButton("valeurs", "Valeur 1");
RadioButton radioButton2 = new RadioButton("valeurs", "Valeur 2");
RadioButton radioButton3 = new RadioButton("valeurs", "Valeur 3");
ClickListener listener = new ClickListener() {
  @Override
  public void onClick(Widget sender) {
     CheckBox checkBox = (CheckBox) sender;
     Window.alert("bouton selectionne = "+checkBox.getText());
  }
};

radioButton1.addClickListener(listener);
radioButton2.addClickListener(listener);
radioButton3.addClickListener(listener);

radioButton2.setValue(true);
VerticalPanel panel = new VerticalPanel();
panel.add(radioButton1);
panel.add(radioButton2);
panel.add(radioButton3);

RootPanel.get("app").add(panel);
```

A partir de GWT version 1.6, la gestion des événements se fait avec des handlers. Plusieurs handlers sont utilisables avec ce composant : BlurHandler, ClickHandler, DomHandler, FocusHandler, KeyDownHandler, KeyPressHandler, KeyUpHandler, MouseDownHandler, MouseMoveHandler, MouseOutHandler, MouseOverHandler, MouseUpHandler et MouseWheelHandler

Exemple :

```
ClickHandler handler = new ClickHandler() {
  @Override
  public void onClick(ClickEvent event) {
    CheckBox cb = (CheckBox) event.getSource();
    if (cb.getValue()) {
      Window.alert("Bonjour");
    }
```

```
        }
    };
    CheckBox bouton = new CheckBox("Afficher");
    bouton.setValue(false);
    bouton.addClickHandler(handler);
    RootPanel.get("app").add(bouton);
```

65.7.2.6. Le composant HyperLink

La suite de cette section sera développée dans une version future de ce document

65.7.3. Les composants de saisie de texte

65.7.3.1. Le composant TextBoxBase

La classe abstraite TextBoxBase encapsule les fonctionnalités de base d'une zone de saisie de texte.

Elle possède plusieurs méthodes pour agir sur le composant dont les principales sont :

Méthodes	Rôle
void cancelKey()	Supprimer un événement clavier reçu par le composant
int getCursorPos()	Obtenir la position courante du curseur dans le texte saisi
String getSelectedText()	Obtenir le texte sélectionné
int getSelectionLength()	Obtenir le nombre de caractères sélectionnés
String getText()	Obtenir le texte
String getValue()	Obtenir la valeur
boolean isReadOnly()	Déterminer si le composant est en lecture seule
void selectAll()	Sélectionner tout le texte
void setCursorPos(int pos)	Positionner le curseur dans le texte
void setReadOnly(boolean readOnly)	Définir si le composant est en lecture seule
void setSelectionRange(int pos, int length)	Définir la portion de texte sélectionnée
void setText(String text)	Initialiser le texte
void setTextAlignment(TextBoxBase.TextAlignConstant align)	Préciser l'alignement du texte
void setValue(String value)	Mettre à jour la valeur du composant sans émettre d'événements
void setValue(String value, boolean fireEvents)	Mettre à jour la valeur du composant

65.7.3.2. Le composant PasswordTextBox

La classe PasswordTextBox encapsule une zone de saisie de texte dont le contenu affiché est masqué. Elle hérite de la classe TextBox.

Exemple :

```
PasswordTextBox textBox = new PasswordTextBox();
textBox.setWidth("200px");
textBox.setMaxLength(10);

RootPanel.get("app").add(textBox);
textBox.setFocus(true);
```

Avant GWT version 1.6, la gestion des événements se fait avec des listeners. Tous les listeners utilisables avec ce composant sont hérités de sa classe mère TextBox.

A partir de GWT version 1.6, la gestion des événements se fait avec des handlers. Tous les handlers utilisables avec ce composant sont hérités de se classe mère TextBox.

65.7.3.3. Le composant TextArea

La classe TextArea encapsule une zone de saisie de texte multilignes. Elle hérite de la classe TextBoxBase.

La méthode setVisibleLines() permet de préciser le nombre de lignes qui seront visibles.

Exemple :

```
TextArea textArea = new TextArea();
textArea.setWidth("200px");
textArea.setVisibleLines(5);
RootPanel.get("app").add(textArea);
```

Avant GWT version 1.6, la gestion des événements se fait avec des listeners. Plusieurs listeners sont utilisables avec ce composant : ChangeListener, ClickListener, FocusListener, KeyboardListener, MouseListener et MouseWheelListener

A partir de GWT version 1.6, la gestion des événements se fait avec des handlers. Plusieurs handlers sont utilisables avec ce composant : DomHandler, ChangeHandler, ValueChangeHandler, BlurHandler, ClickHandler, FocusHandler, KeyDownHandler, KeyPressHandler, KeyUpHandler, MouseDownHandler, MouseMoveHandler, MouseOutHandler, MouseOverHandler, MouseUpHandler et MouseWheelHandler.

65.7.3.4. Le composant TextBox

La classe TextBox encapsule une zone de saisie de texte. Elle hérite de la classe TextBoxBase.

Elle possède plusieurs méthodes pour agir sur le composant dont les principales sont :

Méthodes	Rôle
int getMaxLength()	Obtenir le nombre maximum de caractères saisissables
void setMaxLength(int length)	Limiter le nombre de caractères saisissables

Exemple :

```
TextBox textBox = new TextBox();
textBox.setWidth("200px");
textBox.setText("mon texte");
textBox.setMaxLength(50);

RootPanel.get("app").add(textBox);
textBox.setFocus(true);
textBox.setCursorPos(4);
```

Avant GWT version 1.6, la gestion des événements se fait avec des listeners. Plusieurs listeners sont utilisables avec ce composant : ChangeListener, ClickListener, FocusListener, KeyboardListener, MouseListener et MouseWheelListener

Exemple :

```
TextBox textBox = new TextBox();
textBox.setWidth("200px");
textBox.setMaxLength(10);

textBox.addKeyboardListener(new KeyboardListener() {
  @Override
  public void onKeyDown(Widget sender, char keyCode, int modifiers) {
  }

  @Override
  public void onKeyPress(Widget sender, char keyCode, int modifiers) {
    if (!Character.isDigit(keyCode)) {
      ((TextBox) sender).cancelKey();
    }
  }

  @Override
  public void onKeyUp(Widget sender, char keyCode, int modifiers) {
  }

});

RootPanel.get("app").add(textBox);
textBox.setFocus(true);
```

A partir de GWT version 1.6, la gestion des événements se fait avec des handlers. Plusieurs handlers sont utilisables avec ce composant : DomHandler, ChangeHandler, ValueChangeHandler, BlurHandler, ClickHandler, FocusHandler, KeyDownHandler, KeyPressHandler, KeyUpHandler, MouseDownHandler, MouseMoveHandler, MouseOutHandler, MouseOverHandler, MouseUpHandler et MouseWheelHandler.

Exemple :

```
TextBox textBox = new TextBox();
textBox.setWidth("200px");
textBox.setMaxLength(10);

textBox.addKeyPressHandler(new KeyPressHandler() {
  @Override
  public void onKeyPress(KeyPressEvent event) {
    if (!Character.isDigit(event.getCharCode())) {
      ((TextBox) event.getSource()).cancelKey();
    }
  }
});

RootPanel.get("app").add(textBox);
textBox.setFocus(true) ;
```

65.7.3.5. Le composant RichTextArea

La classe RichTextArea encapsule un composant qui permet à un utilisateur de saisir du texte avec un contenu formater en HTML.

Le composant RichTextArea est un composant évolué mais aussi basique à la fois : il permet le formatage de texte riche par l'utilisateur et propose des objets pour formater le contenu texte mais il ne propose pas d'ensemble de boutons pour faciliter la mise en oeuvre de ces fonctionnalités de formatage.

Selon le navigateur utilisé par l'utilisateur, les fonctionnalités utilisables avec ce composant varient. Ces fonctionnalités sont répaties en trois catégories : Aucune (none), Basique (basic) et Etendue (extended).

Chaque catégorie est encapsulée dans un objet de formatage :

- Basique : encapsulée dans la classe RichTextArea.BasicFormatter
- Ettendue : encapsulée dans la classe RichTextArea.ExtendedFormatter

Pour déterminer si la catégorie est supportée par le navigateur, il faut vérifier qu'il est possible d'obtenir une instance de l'objet correspondant. Il faut utiliser respectivement les méthodes getBasicFormatter() et getExtendedFormatter().

La classe RichTextArea.BasicFormatter propose des méthodes pour des fonctionnalités de formatage basiques notamment :

Méthode	Rôle
String getBackColor()	Obtenir la couleur de fond
String getForeColor()	Obtenir la couleur
boolean isBold()	Indiquer si la zone est en gras
boolean isItalic()	Indiquer si la zone est en italique
boolean isUnderlined()	Indiquer si la zone est soulignée
void selectAll()	Sélectionner tout le texte
void setBackColor(String color)	Modifier la couleur de fond
void setFontName(String name)	Modifier la police de caractères
void setFontSize(RichTextArea.FontSize fontSize)	Modifier la taille de la police de caractères
void setForeColor(String color)	Modifier la couleur
void setJustification(RichTextArea.Justification justification)	Modifier l'alignement
void toggleBold()	Basculer la zone en gras
void toggleItalic()	Basculer la zone en italique
void toggleUnderline()	Basculer la zone en souligné

Toutes ces méthodes agissent sur la sélection courante ou à défaut sur le mot sur lequel le curseur est positionné.

La classe RichTextArea.ExtendedFormatter propose des méthodes pour des fonctionnalités de formatage avancées notamment :

Méthode	Rôle
void createLink(String url)	Créer un hyperlien sur la zone vers l'url fournie en paramètre
void insertHorizontalRule()	Insérer une barre de séparation
void insertImage(String url)	Insérer une image
void insertOrderedList()	Débuter une liste ordonnée

void insertUnorderedList()	Débuter une liste avec puces
boolean isStrikethrough()	Est ce que la zone est barrée
void leftIndent()	Indenter la zone
void removeFormat()	Supprimer tout le formatage de la zone.
void removeLink()	Supprimer l'hyperlien de la zone
void toggleStrikethrough()	Basculer la zone en barré

Exemple :

```
    VerticalPanel panel = new VerticalPanel();
    final RichTextArea richTextBox = new RichTextArea();
    final TextArea textArea = new TextArea();

    final ExtendedFormatter ef = richTextBox.getExtendedFormatter();
    final BasicFormatter bf = richTextBox.getBasicFormatter();
    richTextBox.setWidth("400px");
    richTextBox.setHTML("<h1>Titre</H1><p>Voici
le <b>contenu</b> du paragraphe.</p>");

    textArea.setWidth("400px");
    textArea.setVisibleLines(5);

    HorizontalPanel boutons = new HorizontalPanel();
    panel.add(boutons);
    if (bf != null)
    {
        // ajout des boutons de fomattage basique
        boutons.add(new Button("Gras", new ClickListener()
        {
            public void onClick(Widget sender)
            {
                bf.toggleBold();
            }
        }));

        // ajout des boutons de fomattage ettendu
        if (ef != null)
        {
           boutons.add(new Button("Barre", new ClickListener() {
                public void onClick(Widget sender)
                {
                    ef.toggleStrikethrough();
                }
            }));
        }
    }

    boutons.add(new Button("Text", new ClickListener()
    {
        public void onClick(Widget sender)
        {
            textArea.setText(richTextBox.getText());
        }
    }));
    boutons.add(new Button("Html", new ClickListener()
    {
        public void onClick(Widget sender)
        {
            textArea.setText(richTextBox.getHTML());
        }
    }));

    panel.add(richTextBox);
    panel.add(textArea);

    RootPanel.get("app").add(panel);
    richTextBox.setFocus(true);
```

Avant GWT version 1.6, la gestion des événements se fait avec des listeners. Plusieurs listeners sont utilisables avec ce composant : ClickListener, FocusListener, KeyboardListener et MouseWheelListener

A partir de GWT version 1.6, la gestion des événements se fait avec des handlers. Plusieurs handlers sont utilisables avec ce composant : DomHandler, BlurHandler, ClickHandler, FocusHandler, KeyDownHandler, KeyPressHandler, KeyUpHandler, MouseDownHandler, MouseMoveHandler, MouseOutHandler, MouseOverHandler, MouseUpHandler et MouseWheelHandler

65.7.4. Les composants de sélection de données

65.7.4.1. Le composant ListBox

La classe ListBox encapsule une liste ou une liste déroulante qui permet à l'utilisateur de choisir un ou plusieurs éléments.

La classe ListBox possède plusieurs constructeurs notamment :

Constructeur	Rôle
ListBox()	Constructeur par défaut
ListBox(boolean isMultipleSelect)	Instancier un composant qui autorisera la sélection multiple d'éléments

La classe ListBox possède plusieurs méthodes notamment :

Méthode	Rôle
void addItem(String item)	Ajouter un élément
void addItem(String item, String value)	Ajouter un élément en précisant sa valeur
void clear()	Supprimer tous les éléments
int getItemCount()	Obtenir le nombre d'éléments de la ListBox
String getItemText(int index)	Obtenir le texte de l'élément dont l'index est fourni en paramètre
int getSelectedIndex()	Obtenir l'index de l'élément sélectionné
String getValue(int index)	Obtenir la valeur de l'élément dont l'index est fourni en paramètre
int getVisibleItemCount()	Obtenir le nombre d'éléments visibles
void insertItem(String item, int index)	Insérer un élément à l'index fourni
void insertItem(String item, String value, int index)	Insérer un élément à l'index fourni en précisant la valeur
boolean isItemSelected(int index)	Déterminer si un élément est sélectionné
boolean isMultipleSelect()	Déterminer si la sélection multiple est possible
void removeItem(int index)	Supprimer l'élément dont l'index est fourni en paramètre
void setItemSelected(int index, boolean selected)	Sélectionner ou non l'élément dont l'index est fourni en paramètre
void setItemText(int index, String text)	Modifier le texte de l'élément dont l'index est fourni en paramètre
void setMultipleSelect(boolean multiple)	Activer ou non la sélection multiple. Deprecated : il faut utiliser le constructeur ListBox(boolean)
void setSelectedIndex(int index)	Modifier l'index correspondant à l'élément sélectionné

void setValue(int index, String value)	Modifier la valeur de l'élément dont l'index est fourni en paramètre
void setVisibleItemCount(int visibleItems)	Modifier le nombre d'éléments affichés

L'affichage du composant se fait par défaut sous la forme d'une liste déroulante.

Exemple :

```
    ListBox listBox = new ListBox();

    for (int i=1; i <10 ; i++) {
      listBox.addItem("element "+i);
    }

    RootPanel.get("app").add(listBox);
```

La méthode setVisibleItemCount() permet de préciser le nombre d'éléments de la liste qui sera affiché :

- Avec la valeur 1, le composant est affiché sous la forme d'une liste déroulante
- Avec une autre valeur, le composant est affiché sous la forme d'une liste dont le nombre d'éléments affichés correspond à la valeur fournie

Exemple :

```
    ListBox listBox = new ListBox();

    for (int i=1; i <10 ; i++) {
      listBox.addItem("element "+i);
    }
    listBox.setVisibleItemCount(5);

    RootPanel.get("app").add(listBox);
```

Il est possible de gérer plusieurs événements émis par ce composant.

La classe ListBox possède plusieurs listeners utilisables jusqu'à GWT 1.5 : ChangeListener, ClickListener, MouseListener et MouseWheelListener.

Exemple :

```
    ListBox listBox = new ListBox();

    for (int i=1; i <10 ; i++) {
      listBox.addItem("element "+i, ""+i);
    }
    listBox.addChangeListener(new ChangeListener(){
      @Override
      public void onChange(Widget sender) {
        ListBox list = (ListBox) sender;
        int index = list.getSelectedIndex();
        Window.alert("element selectionne, index="+index+", texte="+list.getItemText(index)+",
valeur="+list.getValue(index));
      }
    } );

    listBox.setVisibleItemCount(5);
    RootPanel.get("app").add(listBox);
```

A partir de GWT 1.6, le composant ListBox propose plusieurs handlers : ClickHandler, MouseDownHandler, MouseMoveHandler, MouseOutHandler, MouseOverHandler, MouseUpHandler et MouseWheelHandler.

Exemple :

```
    ListBox listBox = new ListBox();

    for (int i=1; i <10 ; i++) {
      listBox.addItem("element "+i, ""+i);
    }
    listBox.addChangeHandler(new ChangeHandler(){
      @Override
      public void onChange(ChangeEvent event) {
        ListBox list = (ListBox) event.getSource();
        int index = list.getSelectedIndex();
        Window.alert("element selectionne,
index="+index+", texte="+list.getItemText(index)+",
valeur="+list.getValue(index));
      }
    } );

    listBox.setVisibleItemCount(5);
    RootPanel.get("app").add(listBox);
```

65.7.4.2. Le composant SuggestBox

La classe SuggestBox encapsule une zone de texte qui permet de sélectionner des suggestions dont au moins un des mots commence par le texte saisi.

La classe abstraite SuggestOracle encapsule les suggestions liées à une requête.

L'implémentation par défaut est la classe MultiWordSuggestOracle qui recherche les suggestions parmis celles dont au moins un mot commence par le motif de recherche. La recherche n'est pas sensible à la casse et les suggestions retournées sont triées par ordre alphabétique avec le motif recherché mis en évidence.

La méthode add() permet d'ajouter une suggestion. La méthode addAll() permet d'ajouter plusieurs suggestions et la méthode clear() permet de supprimer toutes les suggestions.

Exemple :

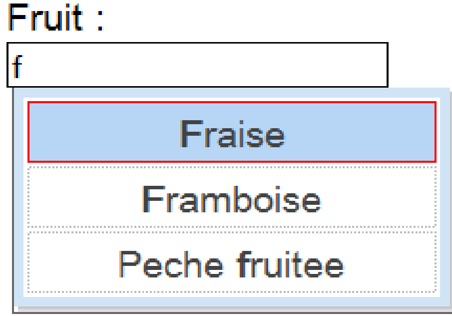
```
    VerticalPanel panel = new VerticalPanel();
    MultiWordSuggestOracle oracle = new MultiWordSuggestOracle();
    oracle.add("Peche fruitee");
    oracle.add("Abricot");
    oracle.add("Banane");
    oracle.add("Fraise");
    oracle.add("Framboise");
    oracle.add("Pomme");
    oracle.add("Poire");

    SuggestBox suggestbox = new SuggestBox(oracle);
    suggestbox.setAnimationEnabled(true);
    panel.add(new Label("Fruit : "));
    panel.add(suggestbox);
    RootPanel.get("app").add(panel);

    suggestbox.setFocus(true);
```

Les principaux styles CSS associés sont :

Style	Rôle
.gwt-SuggestBox	Apparence de la zone de saisie
.gwt-SuggestBoxPopup	Apparence de la popup affichant les suggestions
.gwt-SuggestBoxPopup .item	Apparence d'une suggestion
.gwt-SuggestBoxPopup .item-selected	Apparance de la suggestion sélectionnée

Résultat :

```
/* format de la zone de saisie */
.gwt-SuggestBox
{
border                  :    1px solid #000;
text-align              :    left;
width                   :    200px;
}

/* format de la popup affichant les suggestions */
.gwt-SuggestBoxPopup
{
cursor                  :    pointer;
border                  :    1px solid #666;
border-top              :    0;
background-color        :    #fff;
}

/* format d'une suggestion */
.gwt-SuggestBoxPopup .item
{
text-align              :    center;
border                  :    1px dotted #bbb;
width                   :    200px;
}

/* format de la suggestion selectionnee */
.gwt-SuggestBoxPopup .item-selected
{
border                        :    1px solid #f00;
}
```



Avant GWT version 1.6, la gestion des événements se fait avec des listeners. Plusieurs listeners sont utilisables avec ce composant : ChangeListener, ClickListener, FocusListener et KeyboardListener.

A partir de GWT version 1.6, la gestion des événements se fait avec des handlers. Plusieurs handlers sont utilisables avec ce composant : KeyDownHandler, KeyPressHandler, KeyUpHandler, SelectionHandler, et ValueChangeHandler.

Il est aussi possible d'utiliser plusieurs handlers sur la zone de texte associée au composant dont l'instance est obtenue en invoquant la méthode getTextBox() : ChangeHandler et ClickHandler.

65.7.4.3. Le composant DateBox

La suite de cette section sera développée dans une version future de ce document

65.7.4.4. Le composant DatePicker

La suite de cette section sera développée dans une version future de ce document

65.7.5. Les composants HTML

La suite de cette section sera développée dans une version future de ce document

65.7.5.1. Le composant Frame

La suite de cette section sera développée dans une version future de ce document

65.7.5.2. Le composant HTML

Le composant HTML permet d'afficher un contenu HTML dans l'application. Il propose plusieurs constructeurs dont un qui permet de fournir le code HTML qui sera affiché.

Exemple :

```
HTML widget = new HTML(
  "<div id='panneau' style='background-color: yellow; border:"+
  "1px solid black; width: 200px; text-align: center;'>"+
  "Mon Message d'avertissement</div>");
RootPanel.get("app").add(widget);
```

La méthode setHTML() permet de modifier le contenu du code HTML affiché par le composant.

Il est possible de gérer plusieurs événements émis par ce composant.

La classe HTML possède plusieurs listeners utilisables jusqu'à GWT 1.5 : ClickListener, MouseListener et MouseWheelListener.

A partir de GWT 1.6, le composant HTML propose plusieurs handlers : DomHandler, ClickHandler, ErrorHandler, LoadHandler MouseDownHandler, MouseMoveHandler, MouseOutHandler, MouseOverHandler, MouseUpHandler et

MouseWheelHandler.

65.7.5.3. FileUpload

La suite de cette section sera développée dans une version future de ce document

65.7.5.4. Hidden

La classe Hidden encapsule un champ invisile d'un formulaire HTML.

La classe Hidden possède plusieurs constructeurs notamment :

Constructeur	Rôle
Hidden()	Constructeur par défaut
Hidden(String name)	Constructeur qui attend le nom du champ en paramètre
Hidden(String name, String value)	Constructeur qui attend le nom et la valeur du champ en paramètre

Elle possède plusieurs méthodes qui permettent de manipuler les données de la classe, notamment :

Méthode	Rôle
String getDefaultValue()	Obtenir la valeur par défaut du champ
String getName()	Obtenir le nom du champ
String getValue()	Obtenir la valeur du champ
void setDefaultValue(String defaultValue)	Modifier la valeur par défaut du champ
void setName(name)	Modifier la nom du champ
void setValue(String value)	Modifier la valeur du champ

Ce composant n'ayant pas de rendu graphique, il ne possède aucun événement.

65.7.6. Le composant Tree

Le composant Tree encapsule l'affichage de données sous une forme arborescente. Chaque élément de l'arborescence qui possède au moins un élément fils peut être plier ou déplier pour faire apparaitre ou non les éléments de sa sous arborescence.

La classe Tree propose plusieurs méthodes notamment :

Méthode	Rôle
void add(Widget widget)	Ajouter un élément affichant le composant fourni en paramètre à la racine

	de l'arborescence
TreeItem addItem(String itemText)	Ajouter un élément affichant un texte
void addItem(TreeItem item)	Ajouter un élément à la racine de l'arborescence
TreeItem addItem(Widget widget)	Ajouter un élément qui va afficher le composant fourni en paramètre
void clear()	Supprimer tous les éléments
void ensureSelectedItemVisible()	Rendre visible l'élément sélectionné, au besoin en dépliant ses parents
TreeItem getItem(int index)	Obtenir l'élément dont l'index est fourni
int getItemCount()	Obtenir le nombre d'élément
TreeItem getSelectedItem()	Obtenir l'élément sélectionné
boolean isAnimationEnabled()	Déterminer si l'animation est activité lors du pliement/dépliement d'un élément
java.util.Iterator<Widget> iterator()	Obtenir un iterator sur les composants
void removeItem(TreeItem item)	Supprimer un élément de la racine
void removeItems()	Supprimer tous les éléments de la racine
void setAnimationEnabled(boolean enable)	Activer ou non l'animation lors du pliement ou dépliement d'un élément
void setSelectedItem(TreeItem item)	Sélectionner un élément
java.util.Iterator<TreeItem> treeItemIterator()	Obtenir un itérateur sur les éléments de l'arborescence

La classe Tree est un conteneur pour des éléments de type TreeItem

Le classe TreeItem encapsule un élément d'un composant Tree. Elle propose plusieurs méthodes notamment :

Méthode	Rôle
TreeItem addItem(String itemText)	Ajouter un élément affichant le texte fourni en paramètre
void addItem(TreeItem item)	Ajouter un élément fils
TreeItem addItem(Widget widget)	Ajouter un composant comme élément fils
TreeItem getChild(int index)	Obtenir l'élément fils dont l'index est fourni en paramètre
int getChildCount()	Obtenir le nombre d'éléments fils
int getChildIndex(TreeItem child)	Obtenir l'index d'un élément fils
TreeItem getParentItem()	Obtenir l'élément père
boolean getState()	Déterminer si les éléments fils sont affichés ou non
String getText()	Obtenir le texte de l'élément
Tree getTree()	Obtenir le composant Tree encapsulant toute l'arborescence
Object getUserObject()	Obtenir un objet associé à l'élément
Widget getWidget()	Obtenir le composant associé à l'élément
boolean isSelected()	Déterminer si l'élément est sélectionné ou non
void remove()	Retirer l'élément de l'arborescence
void removeItem(TreeItem item)	Retirer l'élément fils
void removeItems()	Retirer tous les éléments fils

void setSelected(boolean selected)	Sélectionner ou non l'élément
void setState(boolean open)	Afficher ou non les éléments fils
void setText(String text)	Définir le texte de l'élément
void setUserObject(Object userObj)	Associer un objet à l'élément
void setWidget(Widget newWidget)	Définir le composant de l'élément

Les objets de type TreeItem ne peuvent être uniquement rattachés que dans un composant de type Tree.

Exemple :

```
    Tree tree = new Tree();
    tree.setAnimationEnabled(false);

    TreeItem element1 = new TreeItem("Element 1");
    element1.addItem("Element 1-1");
    element1.addItem("Element 1-2");
    element1.addItem("Element 1-3");
    tree.addItem(element1);
    TreeItem sousElement1_3 = new TreeItem("Element 1-3");
    sousElement1_3.addItem("Element 1-3-1");
    sousElement1_3.addItem("Element 1-3-2");
    TreeItem element2 = new TreeItem("Element 2");
    element2.addItem("Element 2-1");
    element2.addItem("Element 2-2");
    TreeItem element2_3 = element2.addItem("Element 2-3");
    tree.addItem(element2);
    element1.addItem(sousElement1_3);

    RootPanel.get("app").add(tree);
```

Il est possible d'afficher un composant dans un élément.

Exemple :

```
    Tree tree = new Tree();
    tree.setAnimationEnabled(false);

    TreeItem element1 = new TreeItem("Element 1");
    element1.addItem("Element 1-1");
    element1.addItem("Element 1-2");
    element1.addItem("Element 1-3");
    tree.addItem(element1);
    TreeItem sousElement1_3 = new TreeItem("Element 1-3");
    sousElement1_3.addItem(new CheckBox("Element 1-3-1"));
    sousElement1_3.addItem(new CheckBox("Element 1-3-2"));
    TextBox element1_3_3 = new TextBox();
    element1_3_3.setText("Element 1-3-3");
    sousElement1_3.addItem(element1_3_3);

    element1.addItem(sousElement1_3);

    RootPanel.get("app").add(tree);
```

Par défaut, le composant Tree ne possède pas de scrollbars : il s'agrandit au fur et à mesure des éléments dépliés. Pour limiter la taille du composant et afficher une scrollbar, il faut l'encapsuler dans un panneau de type ScrollPanel.

Exemple :

```
    Tree tree = new Tree();
    tree.setAnimationEnabled(false);

    TreeItem element1 = new TreeItem("Element 1");
```

```
    element1.addItem("Element 1-1");
    element1.addItem("Element 1-2");
    element1.addItem("Element 1-3");
    tree.addItem(element1);
    TreeItem sousElement1_3 = new TreeItem("Element 1-3");
    sousElement1_3.addItem("Element 1-3-1");
    sousElement1_3.addItem("Element 1-3-2");
    TreeItem element2 = new TreeItem("Element 2");
    element2.addItem("Element 2-1");
    element2.addItem("Element 2-2");
    TreeItem element2_3 = element2.addItem("Element 2-3");
    tree.addItem(element2);
    element1.addItem(sousElement1_3);
    ScrollPanel scrollPanel = new ScrollPanel(tree);
    scrollPanel.setWidth("250px");
    scrollPanel.setHeight("130px");

    RootPanel.get("app").add(scrollPanel);
    tree.setSelectedItem(element2_3);
    tree.ensureSelectedItemVisible();
```

Avant GWT version 1.6, la gestion des événements se fait avec des listeners. Plusieurs listeners sont utilisables avec ce composant : FocusListener, KeyboardListener, MouseListener et TreeListener .

Exemple :

```
        Tree tree = new Tree();
    tree.setAnimationEnabled(false);

    TreeItem element1 = new TreeItem("Element 1");
    element1.addItem("Element 1-1");
    element1.addItem("Element 1-2");
    element1.addItem("Element 1-3");
    tree.addItem(element1);
    TreeItem sousElement1_3 = new TreeItem("Element 1-3");
    sousElement1_3.addItem(new CheckBox("Element 1-3-1"));
    sousElement1_3.addItem(new CheckBox("Element 1-3-2"));
    TextBox element1_3_3 = new TextBox();
    element1_3_3.setText("Element 1-3-3");
    sousElement1_3.addItem(element1_3_3);

    element1.addItem(sousElement1_3);
    tree.addTreeListener(new TreeListener() {
      @Override
      public void onTreeItemSelected(TreeItem item) {
        Window.alert("Selection : "+item.getText());
      }

      @Override
      public void onTreeItemStateChanged(TreeItem item) {
        if (item.getState()) {
          Window.alert("Affichage noeud fils : "+item.getText());
        } else {
          Window.alert("Masquage noeud fils : "+item.getText());
        }
      }
    });

    RootPanel.get("app").add(tree);
```

A partir de GWT version 1.6, la gestion des événements se fait avec des handlers. Plusieurs handlers sont utilisables avec ce composant : DomHandler, BlurHandler, CloseHandler, FocusHandler, KeyDownHandler, KeyPressHandler, KeyUpHandler, MouseDownHandler, MouseMoveHandler, MouseOutHandler, MouseOverHandler, MouseUpHandler, MouseWheelHandler, OpenHandler et SelectionHandler.

Exemple :

```
    Tree tree = new Tree();
```

```
    tree.setAnimationEnabled(false);
    TreeItem element1 = new TreeItem("Element 1");
    element1.addItem("Element 1-1");
    element1.addItem("Element 1-2");
    element1.addItem("Element 1-3");
    tree.addItem(element1);

    TreeItem sousElement1_3 = new TreeItem("Element 1-3");
    sousElement1_3.addItem(new CheckBox("Element 1-3-1"));
    sousElement1_3.addItem(new CheckBox("Element 1-3-2"));
    TextBox element1_3_3 = new TextBox();
    element1_3_3.setText("Element 1-3-3");
    sousElement1_3.addItem(element1_3_3);
    element1.addItem(sousElement1_3);

    tree.addSelectionHandler(new SelectionHandler<TreeItem>() {
      @Override
      public void onSelection(SelectionEvent<TreeItem> event) {
        Window.alert("Selection : " + event.getSelectedItem().getText());
      }
    });

    tree.addOpenHandler(new OpenHandler<TreeItem>() {
      @Override
      public void onOpen(OpenEvent<TreeItem> event) {
        Window.alert("Affichage noeud fils : "
            + event.getTarget().getText());
      }
    });

    tree.addCloseHandler(new CloseHandler<TreeItem>() {
      @Override
      public void onClose(CloseEvent<TreeItem> event) {
        Window.alert("Masquage noeud fils : "
            + event.getTarget().getText());
      }
    });

    RootPanel.get("app").add(tree);
```

65.7.7. Les menus

La classe MenuBar encapsule un menu et les éléments qui le composent.

Exemple :

```
package com.jmdoudoux.testgwt.client;

import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.user.client.Command;
import com.google.gwt.user.client.Window;
import com.google.gwt.user.client.ui.MenuBar;
import com.google.gwt.user.client.ui.RootPanel;

public class MonApp implements EntryPoint {

  public void onModuleLoad() {
    MenuBar menu = new MenuBar();
    MenuBar menu1 = new MenuBar(true);
    MenuBar menu2 = new MenuBar(true);

    menu2.addItem("menu2_1", new MonMenuCommand());
    menu2.addItem("menu2_2", new MonMenuCommand());
    menu2.addItem("menu2_3", new MonMenuCommand());

    menu1.addItem("menu1_1", new MonMenuCommand());
    menu1.addItem("menu1_2", new MonMenuCommand());
```

```
    menu.addItem("menu1", menu1);
    menu.addItem("menu2", menu2);
    menu.setAutoOpen(true);
    RootPanel.get("menu").add(menu);

    menu1.addStyleName("submenu");
    menu2.addStyleName("submenu");
  }

  public class MonMenuCommand implements Command {
    public void execute() {
      Window.alert("Element du menu clické");
    }
  }
}
```

Le rendu du menu est assuré par des styles CSS. Le plus simple est de les définir dans un fichier .css

Résultat :

```
body {
   margin: 0;
   padding: 0;
   background: #ffffff;
}

.gwt-MenuBar {
   background: #a0a0a0;
   border: 1px solid #3f3f3f;
   cursor: pointer;
}

.gwt-MenuBar .gwt-MenuItem {
   font-family: Arial, sans-serif;
   font-size: 12px;
   color: #ffffff;
   font-weight: bold;
   padding-right: 10px;

}

.gwt-MenuBar .gwt-MenuItem-selected {
   font-family: Arial, sans-serif;
   font-size: 12px;
   color: #ffffff;
   font-weight: bold;
   padding-right: 10px;
}
```

Ce fichier doit être déclaré dans le fichier de configuration de l'application grâce au tag <stylesheet>. L'attribut src permet de préciser le fichier.

Exemple :

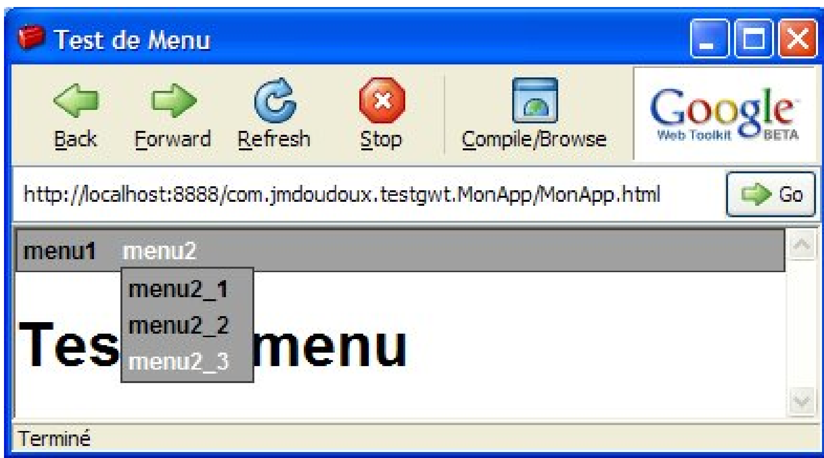
```
<module>

    <!-- Inherit the core Web Toolkit stuff.                  -->
    <inherits name='com.google.gwt.user.User' />

    <stylesheet src="MonApp.css" />

    <!-- Specify the app entry point class.                   -->
    <entry-point class='com.jmdoudoux.testgwt.client.MonApp' />

</module>
```

Enfin, un calque nommé menu est défini dans le fichier html de la page.

Exemple :

```
<html>
<head>
<title>Test de Menu</title>
<style>
  body,td,a,div,.p{font-family:arial,sans-serif}
  div,td{color:#000000}
  a:link,.w,.w a:link{color:#0000cc}
  a:visited{color:#551a8b}
  a:active{color:#ff0000}
</style>

<meta name='gwt:module' content='com.jmdoudoux.testgwt.MonApp'>
</head>
<body>
<div id="menu"></div>
<script language="javascript" src="gwt.js"></script>
<iframe id="__gwt_historyFrame" style="width:0;height:0;border:0"></iframe>
<h1>Test de menu</h1>
</body>
</html>
```

Résultat :



65.7.8. Le composant TabBar

La classe TabBar encapsule une barre d'onglets. Ce composant est essentiellement utilisé dans un panneau TabPanel

Exemple :

```
    VerticalPanel panel = new VerticalPanel();
    final TabBar tabBar = new TabBar();
    final Label label = new Label();

    tabBar.addTab("Onglet 1");
    tabBar.addTab("Onglet 2");
    tabBar.addTab("Onglet 3");
    tabBar.addTab("Onglet 4");
    tabBar.addTabListener(new TabListener() {
      @Override
      public boolean onBeforeTabSelected(SourcesTabEvents sender, int tabIndex) {
        return true;
      }

      @Override
      public void onTabSelected(SourcesTabEvents sender, int tabIndex) {
```

```
        int selectionne = tabBar.getSelectedTab();

        if (selectionne != 0) {
          tabBar.setTabEnabled(3, true);
        }

        if (tabIndex == 0) {
          tabBar.setTabEnabled(3, false);
        }

        label.setText("Selection de l'onglet : "+tabBar.getTabHTML(tabIndex));
      }
    });

    panel.add(tabBar);
    panel.add(label);
    tabBar.selectTab(0);

    RootPanel.get("app").add(panel);
```

Avant GWT version 1.6, la gestion des événements se fait avec des listeners. Un seul listener est utilisable avec ce composant : TabListener.

Exemple :

```
    VerticalPanel panel = new VerticalPanel();
    final TabBar tabBar = new TabBar();
    final Label label = new Label();

    tabBar.addTab("Onglet 1");
    tabBar.addTab("Onglet 2");
    tabBar.addTab("Onglet 3");
    tabBar.addTab("Onglet 4");
    tabBar.addTabListener(new TabListener() {
      @Override
      public boolean onBeforeTabSelected(
          SourcesTabEvents sender, int tabIndex)
      {
        return true;
      }

      @Override
      public void onTabSelected(SourcesTabEvents sender, int tabIndex)
      {
        int selectionne = tabBar.getSelectedTab();

        if (selectionne != 0) {
          tabBar.setTabEnabled(3, true);
        }

        if (tabIndex == 0) {
          tabBar.setTabEnabled(3, false);
        }

        label.setText("Selection de l'onglet : "+tabBar.getTabHTML(tabIndex));
      }
    });
    panel.add(tabBar);
    panel.add(label);
    tabBar.selectTab(0);

    RootPanel.get("app").add(panel);
```

A partir de GWT version 1.6, la gestion des événements se fait avec des handlers. Plusieurs handlers sont utilisables avec ce composant : SelectionBeforeHandler et SelectionHandler.

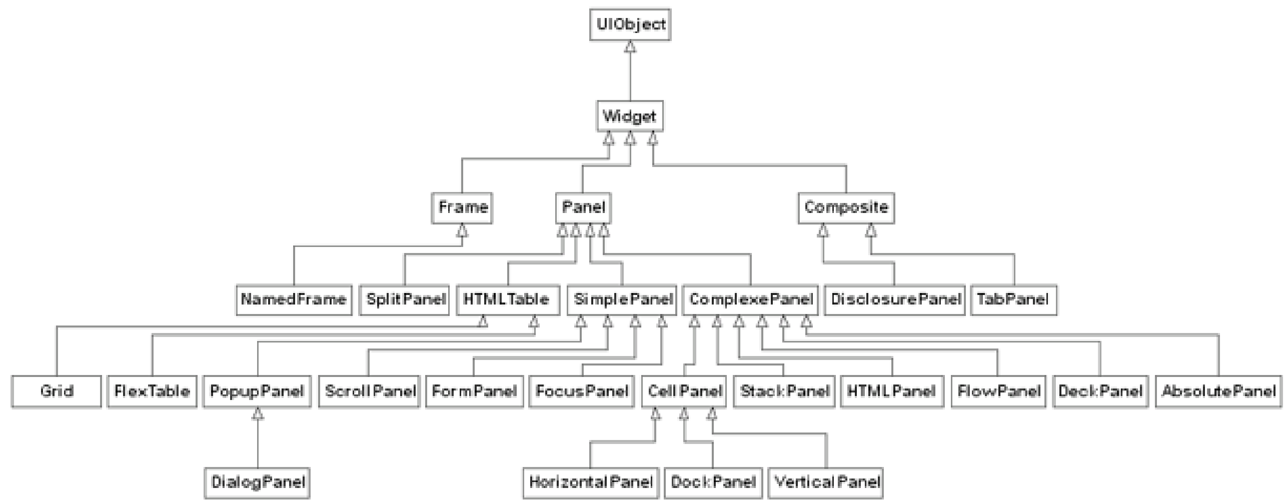
Exemple :

```
    VerticalPanel panel = new VerticalPanel();
    final TabBar tabBar = new TabBar();
    final Label label = new Label();

    tabBar.addTab("Onglet 1");
    tabBar.addTab("Onglet 2");
    tabBar.addTab("Onglet 3");
    tabBar.addTab("Onglet 4");
    tabBar.addSelectionHandler(new SelectionHandler<Integer>() {
      @Override
      public void onSelection(SelectionEvent<Integer> event) {
        int courant = tabBar.getSelectedTab();
        int selectionne = event.getSelectedItem();

        if (courant != 0) {
          tabBar.setTabEnabled(3, true);
        }

        if (selectionne == 0) {
          tabBar.setTabEnabled(3, false);
        }

        label.setText("Selection de l'onglet : "+tabBar.getTabHTML(selectionne));
      }
    });

    panel.add(tabBar);
    panel.add(label);
    tabBar.selectTab(0);

    RootPanel.get("app").add(panel);
```

65.8. Les panneaux (panels)

Ils permettent d'organiser les composants affichés sur la page : selon leurs fonctionnalités, ils peuvent gérer le positionnement des composants ou leur visibilité.

Les panneaux permettent d'assurer la structure visuelle de l'application. GWT propose un ensemble complet de composants graphiques de type conteneur pour organiser et assembler les composants graphiques.



Les composants de type Panel contiennent d'autres composants et permettent de les organiser. Ils sont donc plus que des conteneurs car ils sont aussi des gestionnaires de positionnement.

Ceci est essentiellement due au fait que le rendu HTML de ces composants est généralement soit un élément table ou div HTML.

Panneau	Rendu HTML	Version de GWT	Description
AbsolutePanel	DIV	1.0	Panneau qui permet le positionnement absolu (grâce à leur position) des composants
CaptionPanel		1.5	Panneau qui possède un titre
CellPanel	TABLE	1.0	Panneau abstrait pour une cellule d'un panneau qui en est composé
ComplexPanel		1.0	Classe abstraite de base pour les panneaux possédant plusieurs composants
DeckPanel	DIV		Panneau qui n'affiche qu'un seul composant à la fois parmis ceux qu'il contient
DisclosurePanel	TABLE	1.4	
DockPanel	TABLE	1.0	Panneau qui permet de positionner les composants dans 5 zones (N, S, E, W et centre)
FlexTable		1.0	
FlowPanel	DIV	1.0	Panneau qui est un simple DIV
FocusPanel	DIV	1.0	
FormPanel	DIV	1.1	Panneau qui contient un formulaire HTML
Frame	IFRAME	1.0	Panneau sous la forme d'un IFRAME
Grid		1.0	
HorizontalPanel	TABLE	1.0	Panneau avec alignement horizontal des composants
HorizontaSplitlPanel	DIV	1.4	Panneau composé de deux cellules l'une à côté de l'autre, redimensionnable en hauteur
HTMLPanel	DIV	1.0	Panneau qui permet d'afficher un contenu HTML
HTMLTable			
LazyPanel	DIV	1.6	Panneau qui permet de différer la création de son rendu au moment de son affichage (GWT 1.6)
Panel		1.0	Classe abstraite de base pour les autres panels
PopupPanel			
RootPanel		1.0	
ScrollPanel	DIV	1.0	
SimplePanel	DIV	1.0	Classe abstraite de base pour les panneaux ne possédant qu'un seul composant
StackPanel	TABLE	1.0	
TabPanel	TABLE	1.0	Panneau sous la forme d'onglets
VerticalPanel	TABLE	1.0	Panneau avec alignement vertical des composants
VerticaSplitlPanel	DIV	1.4	Panneau composé de deux cellules l'une au dessus de l'autre, redimensionnable en hauteur

Comme pour les composants, ils possèdent une représentation en Java et en Javascript. Le constructeur de la classe se charge de créer le ou les éléments nécessaires dans l'arbre DOM.

Chaque panneau peut contenir des composants ou d'autres panneaux. Le panneau principal est encapsulé dans la classe RootPanel.

Les autres panneaux servent de conteneur pour les composants graphiques.

La suite de cette section sera développée dans une version future de ce document

65.8.1. La classe Panel

La plupart des panneaux qui contiennent un ou plusieurs composants héritent de façon directe ou indirecte de la classe Panel.

Cette classe implémente l'interface HasWidgets. Cette interface définit plusieurs méthodes :

Méthode	Rôle
add(Widget)	Ajouter le composant fourni en paramètres
Clear()	Supprimer tous les composants contenus dans le panneau
iterator()	Obtenir un objet de type Iterator pour parcourir tous les composants inclus dans le panneau.
remove(Widget)	Supprimer le composant fourni en paramètre

Cette classe possède plusieurs classes filles directes : ComplexPanel, HorizontalSplitPanel, HTMLTable, SimplePanel, et VerticalSplitPanel

65.8.2. La classe RootPanel

Cette classe encapsule la page affichée dans le navigateur : elle permet donc d'associer l'application au navigateur. C'est le seul panneau qui possède un accès à la page du navigateur.

La classe RootPanel est la classe qui encapsule le panneau racine qui doit obligatoirement être au sommet de hiérarchie des panneaux et composants : ce panneau est donc toujours au sommet de la hiérarchie des panneaux de la page de l'application.

Ce panneau encapsule une partie de la page html de l'application en permettant un accès à certains de ses éléments : elle n'est donc pas un conteneur au sens strict mais elle permet un accès aux éléments de l'arbre DOM de la page.

La méthode get() permet d'obtenir une référence sur l'élément du DOM dont l'id est fourni en paramètre de la méthode. Sans paramètre, cette méthode renvoie l'objet de type RootPanel qui encapsule la page.

Exemple :

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN">
<html>
  <head>
    <meta
http-equiv="content-type" content="text/html;
charset=ISO-8859-15">
    <link
type="text/css" rel="stylesheet" href="TestGWT.css">

<title>Application de test GWT</title>
    <script
type="text/javascript" language="javascript"
src="testgwt/testgwt.nocache.js"></script>
```

```
  </head>
  <body>

<h1>Application de test GWT</h1>
    <table
align="center">
      <tr>
        <td
id="menu"></td><td id="app"></td>
      </tr>
    </table>
  </body>
</html>
```

Il n'est pas possible d'instancier un objet de type RootPanel : il faut utiliser la méthode get() qui renvoie le panneau par défaut ou sa surcharge qui attend en paramètre l'id d'un élément de la page html de l'application.

Exemple :

```
    Label libelle = new Label("Bonjour");

    RootPanel.get("app").add(libelle);
```

Il est ainsi possible d'accéder à tous les éléments de la page html qui possède un id.

65.8.3. La classe SimplePanel

La classe SimplePanel est un panneau qui peut contenir un seul composant.

Ce panneau est implémenté sous la forme d'un simple tag DIV en HTML.

Exemple :

```
    SimplePanel panel = new SimplePanel();
    panel.setSize("200px", "50px");
    panel.addStyleName("monPanneau");
    Label label = new Label("Contenu du panneau");
    panel.add(label);
    RootPanel.get("app").add(panel);
```

Résultat :

```
.monPanneau {
border-color: black;
border-width : 1px;
border-style: solid;
background-color: silver;
text-align: center;
}
```

Contenu du panneau

65.8.4. La classe ComplexPanel

La classe abstraite ComplexPanel est la classe de base pour un panneau qui peut contenir plusieurs composants.

Elle propose plusieurs méthodes de base pour gérer les composants contenus dans le panneau notamment :

Méthode	Rôle
add(Widget, Element);	Ajouter un nouveau composant
getChildren()	Obtenir une collection des composants contenus dans le panneau
getWidget(int)	Obtenir le composant à partir de son index
getWidgetCount()	Obtenir le nombre de composants contenus dans le panneau
getWidgetIndex(Widget)	Obtenir l'index d'un composant
insert(Widget, Element, int)	Insérer un nouveau composant à l'index fourni
iterator()	Obtenir un iterator sur les composants du panneau
remove(int)	Supprimer le composant à l'index fourni
remove(Widget)	Supprimer le composant

La classe ComplexPanel possède plusieurs classes filles notamment : AbsolutePanel, CellPanel, DeckPanel, FlowPanel, HTMLPanel, et StackPanel.

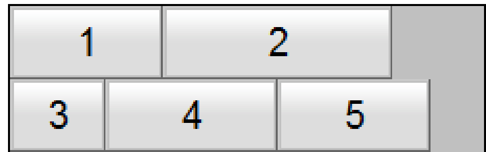
65.8.5. La classe FlowPanel

La classe FlowPanel est un panneau dans lequel les composants sont ajoutés les uns à la suite des autres avec un passage à la ligne dès que la place manque pour contenir le composant. Ce panneau arrange les composants qu'il contient les uns à côté des autres en allant du haut à gauche vers le bas à droite.

Le panneau FlowPanel est un simple élément HTML <DIV> avec le style display:inline dans l'arbre DOM .

Exemple :

```
    FlowPanel panel = new FlowPanel();
    panel.setWidth("250px");
    panel.addStyleName("monPanneau");

    Button bouton = new Button("1");
    bouton.setWidth("80px");
    panel.add(bouton);

    bouton = new Button("2");
    bouton.setWidth("120px");
    panel.add(bouton);

    bouton = new Button("3");
    bouton.setWidth("50px");
    panel.add(bouton);

    bouton = new Button("4");
    bouton.setWidth("90px");
    panel.add(bouton);

    bouton = new Button("5");
    bouton.setWidth("80px");
    panel.add(bouton);

    RootPanel.get("app").add(panel);
```



65.8.6. La classe DeckPanel

La classe DeckPanel est un panneau qui affiche un composant à la fois. Ce panneau est utilisé dans un TabPanel pour afficher le contenu de l'onglet sélectionné.

Un exemple typique d'utilisation est pour la mise en oeuvre d'un assistant.

Exemple :

```
final DeckPanel panel = new DeckPanel();
panel.setSize("200px", "50px");
panel.addStyleName("monPanneau");
panel.add(new Label("Contenu cellule 1"));
panel.add(new Label("Contenu cellule 2"));
panel.add(new Label("Contenu cellule 3"));

panel.showWidget(0);

Timer timer = new Timer()
{
  public void run()
  {
    int index = panel.getVisibleWidget();
    index++;
    if (index == panel.getWidgetCount()) {
      index = 0;
    }

    panel.showWidget(index);
  }
};

timer.scheduleRepeating(2000);

RootPanel.get("app").add(panel);
```

65.8.7. La classe TabPanel

La classe TabPanel est un panneau composé d'onglets. Il est composé d'un ensemble de boutons, un pour chaque onglet et d'un composant DeckPanel qui affiche le contenu de l'onglet sélectionné.

Exemple :

```
StringBuffer sb = new StringBuffer("<p>");
TabPanel panel = new TabPanel();
Panel contenu;
contenu = new SimplePanel();
contenu.add(new Label("Contenu du panneau 1"));
panel.add(contenu, "Onglet 1");
contenu = new SimplePanel();
contenu.add(new Label("Contenu du panneau 2"));
panel.add(contenu, "Onglet 2");
contenu = new SimplePanel();
contenu.add(new Label("Contenu du panneau 3"));
panel.add(contenu, "Onglet 3");
panel.selectTab(0);
panel.setSize("500px", "250px");
```

```
    RootPanel.get("app").add(panel);
```

Onglet 1 Onglet 2 Onglet 3

Contenu du panneau 3

65.8.8. La classe FocusPanel

La suite de cette section sera développée dans une version future de ce document

65.8.9. La classe HTMLPanel

La classe HTMLPanel est un composant qui peut afficher du code HTML.

La classe HTMLPanel permet d'ajouter et de supprimer des éléments à partir de leur id.

Exemple :

```
    String html = "<table><tr><td nowrap><div id='libelle'>"
        + "</div></td><td><div id='saisie'>"
        + "</div></td></tr></table>";
    HTMLPanel panel = new HTMLPanel(html);

    panel.setSize("250px", "120px");
    panel.add(new Label("Libelle a saisir :"), "libelle");
    panel.add(new TextBox(), "saisie");

    RootPanel.get("app").add(panel);
```

Libelle a saisir :

65.8.10. La classe FormPanel

La suite de cette section sera développée dans une version future de ce document

65.8.11. La classe CellPanel

Cette classe est la classe abstraite pour une cellule d'un panneau qui en est composé. C'est la super classe de plusieurs panneaux : DockPannel, HorizontalPannel et VerticalPannel. Tous ces panneaux organisent les composants qu'ils contiennent dans des cellules.

65.8.12. La classe DockPanel

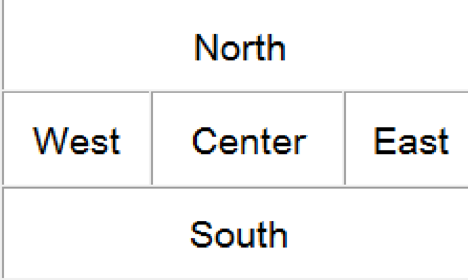
La classe DockPanel encapsule un panneau découpé en cinq parties positionnées relativement à la partie centrale :

- Une ligne qui contient la partie nord (NORTH)
- Une ligne qui contient les parties ouest (WEST), centre (CENTER) et est (EAST)
- Une ligne qui contient la partie sud (SOUTH)

Ce panneau utilise une table HTML.

Exemple :

```
final DockPanel panel = new DockPanel();
panel.setVerticalAlignment(HasAlignment.ALIGN_MIDDLE);
panel.setHorizontalAlignment(HasAlignment.ALIGN_CENTER);
panel.setWidth("250px");
panel.setHeight("150px");
panel.setBorderWidth(1);

panel.add(new Label("North"), DockPanel.NORTH);
panel.add(new Label("South"), DockPanel.SOUTH);
panel.add(new Label("West"), DockPanel.WEST);
panel.add(new Label("East"), DockPanel.EAST);
panel.add(new Label("Center"), DockPanel.CENTER);
```



65.8.13. La classe HorizontalPanel

La classe HorizontalPanel encapsule un panneau qui peut contenir plusieurs composants alignés les uns à côté des autres.

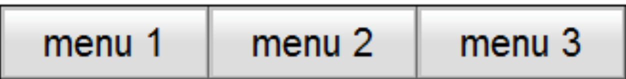
Concrètement, c'est un tableau HTML où chaque composant est inséré dans une nouvelle cellule placée horizontalement à côté de la précédente.

Attention, ce panneau n'est visible que s'il contient au moins un composant même si sa taille est définie.

Exemple :

```
HorizontalPanel panel = new HorizontalPanel();
panel.addStyleName("monPanneau");

Button menu1 = new Button("menu 1");
Button menu2 = new Button("menu 2");
Button menu3 = new Button("menu 3");
```

```
    panel.add(menu1);
    panel.add(menu2);
    panel.add(menu3);

    RootPanel.get("app").add(panel);
```



65.8.14. La classe VerticalPanel

La classe VerticalPanel encapsule un panneau qui peut contenir plusieurs composants alignés les uns au dessus des autres. Ce panneau arrange les composants de façon verticale, les uns en dessous des autres comme dans une colonne.

Le panneau VerticalPanel est un élément HTML <TABLE> dans l'arbre DOM. A chaque appel de la méthode add(), une cellule est ajoutée dans une nouvelle ligne du tableau. Cette cellule contient le composant en paramètre de la méthode.

Exemple :

```
    VerticalPanel panel = new VerticalPanel();
    panel.addStyleName("monPanneau");

    Button menu1 = new Button("menu 1");
    Button menu2 = new Button("menu 2");
    Button menu3 = new Button("menu 3");

    panel.add(menu1);
    panel.add(menu2);
    panel.add(menu3);

    RootPanel.get("app").add(panel);
```

Attention, ce panneau n'est visible que s'il contient au moins un composant même si sa taille est définie.

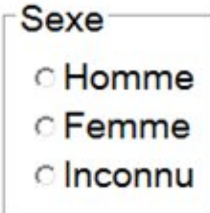
65.8.15. La classe CaptionPanel

Ce panneau possède un titre : il permet de grouper des composants appartenant à un même ensemble fonctionnel.

Exemple :

```
    CaptionPanel panel = new CaptionPanel("Sexe");
    VerticalPanel vpBoutons = new VerticalPanel();
    vpBoutons.setStyleName("sexes");

    RadioButton rbHomme = new RadioButton("sexe","Homme");
    vpBoutons.add(rbHomme);
    RadioButton rbFemme = new RadioButton("sexe","Femme");
    vpBoutons.add(rbFemme);
    RadioButton rbInconnu = new RadioButton("sexe","Inconnu");
    vpBoutons.add(rbInconnu);
```

```
  panel.setContentWidget(vpBoutons);
```



65.8.16. La classe PopupPanel

La classe PopupPanel encapsule un panneau qui est capable de s'afficher au dessus de tous les autres composants.

Il est possible que le panneau s'efface automatiquement (auto-hide) dès que l'utilisateur clique en dehors du panneau. Le panneau peut aussi être modal.

Ce panneau peut avoir de nombreuses utilités : afficher des données, demander une confirmation ou une petite quantité de données, verrouiller l'application, ...

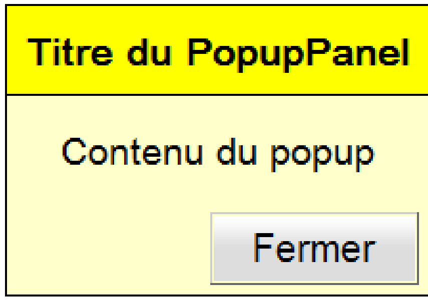
Exemple :

```
private static class TestPopupPanel extends PopupPanel {

  public TestPopupPanel(String message) {
    super(true, true);

    this.setStyleName("demo-popup");

    VerticalPanel contenuPopupPanel = new VerticalPanel();

    this.setAnimationEnabled(true);
    HTML titre = new HTML("Titre du PopupPanel");

    titre.setStyleName("demo-popup-header");
    HTML contenu = new HTML(message);

    contenu.setStyleName("demo-popup-message");

    // bouton pour fermer le popup

    ClickListener listener = new ClickListener() {
        public void onClick(Widget sender)
        {
          hide();
        }
    };
    Button boutonFermer = new Button("Fermer", listener);
    SimplePanel holder = new SimplePanel();
    holder.add(boutonFermer);

    holder.setStyleName("demo-popup-footer");
    contenuPopupPanel.add(titre);
    contenuPopupPanel.add(contenu);
    contenuPopupPanel.add(holder);

    this.setWidget(contenuPopupPanel);
  }

  public void onModuleLoad() {
    final TestPopupPanel popup = new TestPopupPanel("Contenu du popup");

    ClickListener listener = new ClickListener()
    {
      public void onClick(Widget sender)
```

```
        {
          popup.center();
        }
    };
    Button bouton = new Button("Afficher", listener);

    RootPanel.get("app").add(bouton);
  }
```

Résultat :

```
.demo-popup
{
background-color        :     #ffc;
border                  :     1px solid #000;
}

.demo-popup-header
{
background-color        :     #ff0;
font-weight             :     bold;
border-bottom           :     1px solid #000;
padding                 :     10px;
}

.demo-popup-message
{
padding                 :     15px;
text-align              :     center;
}

.demo-popup-footer
{
padding                 :     5px;
text-align              :     right;
width                   :     100%;
}
```



Cet exemple est purement éducatif car une partie de ses fonctionnalités est implémentée dans le composant DialogBox.

L'instance du panneau n'a pas besoin d'être rattachée au RootPanel ou à tout autre composant.

Les méthodes center() ou show() permettent d'afficher le panneau et la méthode hide() permet de le masquer.

La taille du panneau est déterminée selon la taille du composant qu'il contient.

65.8.17. La classe DialogBox

La suite de cette section sera développée dans une version future de ce document

65.8.18. La classe DisclosurePanel

La classe DisclosurePanel est un panneau composé de deux parties : une en-tête toujours visible et une partie principale qu'il est possible de masquer ou afficher en cliquant sur l'en-tête.

Ce composant est pratique lorsqu'il y a beaucoup de données à afficher.

Exemple :

```
final DisclosurePanel panel = new DisclosurePanel("Cliquez pour ouvrir");

panel.addEventHandler(new DisclosureHandler() {
  public void onClose(DisclosureEvent event) {
    panel.getHeaderTextAccessor().setText("Cliquez pour ouvrir");
  }

  public void onOpen(DisclosureEvent event) {
    panel.getHeaderTextAccessor().setText("Cliquez pour fermer");
  }
});

panel.add(new Image("images/logo_java.jpg"));

panel.setWidth("300px");
RootPanel.get("app").add(panel);
```

Il est possible de personnaliser l'en-tête en dérivant d'un panneau par exemple HorizontalPanel et en fournissant une instance de cette classe à la méthode setHeader().

65.8.19. La classe AbsolutePanel

Ce panneau permet le positionnement absolu (grâce à des coordonnées) des composants dans le panneau. Le rendu de ce panneau en HTML est un div.

La taille du panneau n'est pas automatiquement agrandie lors de l'ajout de composant hors de sa surface d'affichage définie par sa taille.

Exemple :

```
AbsolutePanel panel = new AbsolutePanel();
panel.setSize("250px", "100px");
panel.setStyleName("monPanneau");
Label label1 = new Label("Mon premier texte");

panel.add(label1, 50, 30);
Label label2 = new Label("Mon second texte");
panel.add(label2, 65, 45);

RootPanel.get("app").add(panel);
```

65.8.20. La classe StackPanel

La classe StackPanel encapsule un composant qui contient plusieurs sous panneaux possédant un titre dont un seul peut être affiché .

Exemple :

```
StackPanel panel = new StackPanel();
Label label;
label = new Label("Contenu 1");
panel.add(label, "Titre 1", false);
label = new Label("Contenu 2");
panel.add(label, "Titre 2", false);
label = new Label("Contenu 3");
panel.add(label, "Titre 2", false);
panel.setSize("200px", "200px");
RootPanel.get("app").add(panel);
```

65.8.21. La classe ScrollPanel

La classe ScrollPanel est un panneau qui peut contenir un seul composant et qui possède une barre de défilement.

Exemple :

```
StringBuffer sb = new StringBuffer("<p>");
for (int i=0; i<10; i++) {
  sb.append("ligne de test ... <br>");
}

sb.append("</p>");
ScrollPanel panel = new ScrollPanel(new HTML(sb.toString()));

panel.setSize("200px", "100px");

RootPanel.get("app").add(panel);
```

ligne de test ...
ligne de test ...
ligne de test ...
ligne de test ...

65.8.22. La classe FlexTable

La classe FlexTable encapsule un panneau qui est une table dont le nombre de cellule peut varier pour chaque ligne. A sa création, une FlexTable n'a pas de taille explicite.

Ce panneau utilise une table HTML.

L'index de la première cellule vaut 0.

Exemple :

```
Personne[] personnes = new Personne[] {
    new Personne(1, "Nom 1", "Prenom 1", 171),
    new Personne(2, "Nom 2", "Prenom 2", 172),
    new Personne(3, "Nom 3", "Prenom 3", 173) };
FlexTable t = new FlexTable();

t.setTitle("Personnes");
t.setText(0, 0, "Id");
t.setText(0, 1, "Nom");
t.setText(0, 2, "Prenom");
t.setText(0, 3, "Taille");

t.setCellPadding(5);
t.setCellSpacing(0);

t.setBorderWidth(1);
for (int i = 0; i < 4; i++) {
  t.getColumnFormatter().addStyleName(i, "monPanneau");
}
for (int i = 0; i < personnes.length; i++) {
  Personne personne = personnes[i];
  t.setText(i + 1, 0, "" + personne.getId());
  t.setText(i + 1, 1, personne.getNom());
  t.setText(i + 1, 2, personne.getPrenom());
  t.setText(i + 1, 3, "" + personne.getTaille());
}

RootPanel.get("app").add(t);
```

Id	Nom	Prenom	Taille
1	Nom 1	Prenom 1	171
2	Nom 2	Prenom 2	172
3	Nom 3	Prenom 3	173

La méthode setColSpan() de la classe FlexCellFormater permet de fusionner deux cellules.

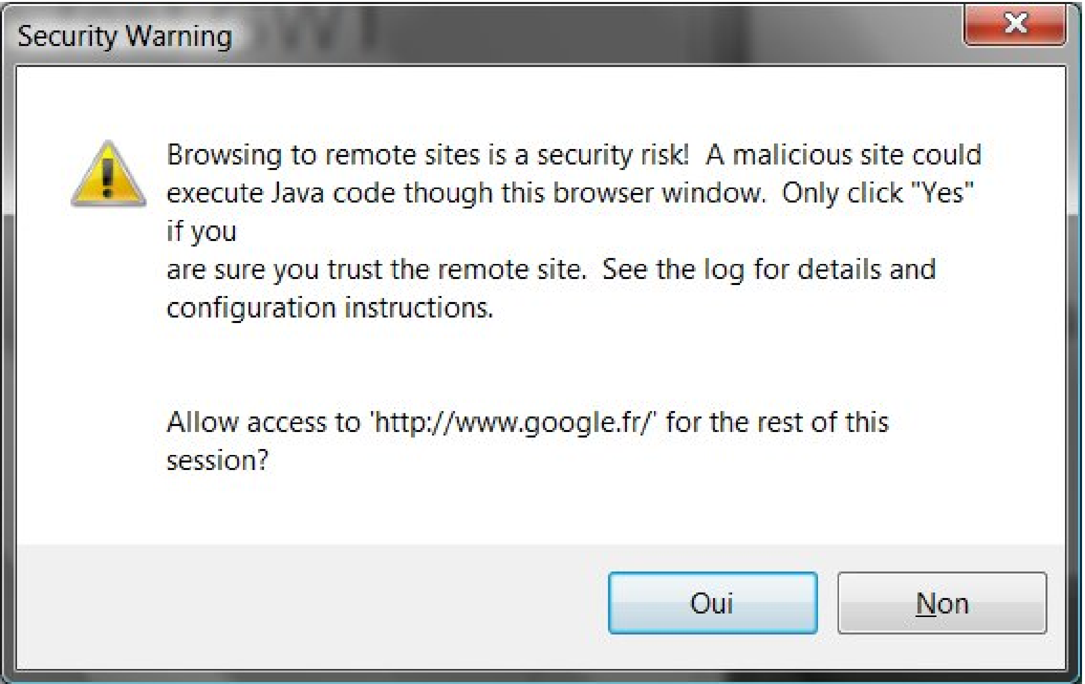
65.8.23. La classe Frame

Ce composant encapsule un IFrame HTML.

Exemple :

```
Frame frame = new Frame("http://www.google.fr/");
frame.setWidth("600px");
frame.setHeight("350px");

RootPanel.get("app").add(frame);
```

Les Iframes sont fréquemment utilisés pour effectuer des opérations d'un site distant à l'insu de l'utilisateur. Lors de l'affichage de l'application utilisant un IFrame un message d'avertissement est affiché en demandant la confirmation de l'accès au site.



65.8.24. La classe Grid

Ce composant encapsule un tableau HTML : c'est donc une grille composée de cellules.

Il faut définir le nombre de cellules de la grille (nombre de colonnes et de lignes) avant de pouvoir insérer un composant dans une cellule.

L'ajout d'un composant dans une cellule se fait en utilisant la méthode setWidget().

Exemple :

```
Grid grille = new Grid(3, 3);
grille.setSize("250px", "100px");

for (int i = 0; i < 3 ; i++ ) {
  for (int j = 0; j < 3 ; j++ ) {
    grille.setWidget(i, j, new Label("Libelle "+(i+j)));
  }
}

RootPanel.get("app").add(grille);
```

La méthode setText() permet de facilement remplir une cellule de la grille avec du texte. L'exemple ci-dessous est identique au précédent.

Exemple :

```
Grid grille = new Grid(3, 3);
grille.setSize("250px", "100px");

for (int i = 0; i < 3 ; i++ ) {
  for (int j = 0; j < 3 ; j++ ) {
    grille.setText(i, j, "Libelle "+(i+j));
  }
}

RootPanel.get("app").add(grille);
```

Libelle 0 Libelle 1 Libelle 2
Libelle 1 Libelle 2 Libelle 3
Libelle 2 Libelle 3 Libelle 4

Si la cellule à remplir est en dehors de la taille définie dans le constructeur alors une exception de type IndexOutOfBoundsException est levée.

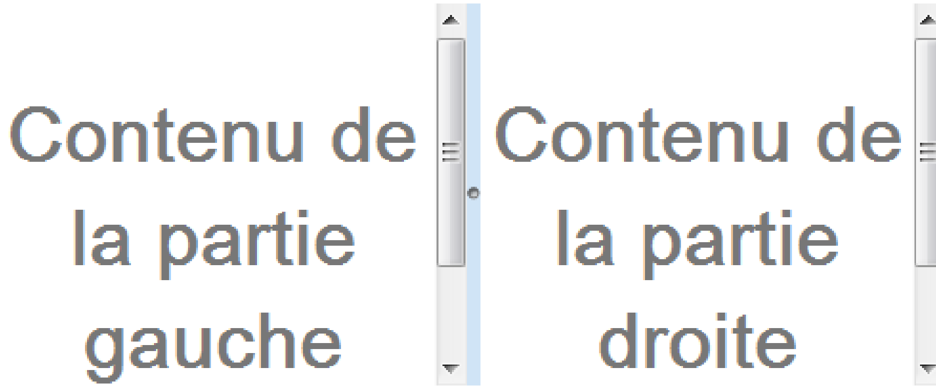
Il est possible de redimensionner le nombre de cellules de la grille grâce aux méthodes resize(), resizeColumns() et resizeRows().

65.8.25. La classe HorizontalSplitPanel

La classe HorizontalSplitPanel encapsule un panneau composé de deux cellules l'une à côté de l'autre. La taille des cellules est adaptable l'une au détriment de l'autre. Une barre de défilement est affichée pour une cellule si sa taille est trop petite pour afficher son contenu.

Exemple :

```
HorizontalSplitPanel panel = new HorizontalSplitPanel();
panel.setSize("500px", "200px");
panel.setLeftWidget(new HTML("<H1>Contenu de la partie gauche</H1>"));
panel.setRightWidget(new HTML("<H1>Contenu de la partie droite</H1>"));

RootPanel.get("app").add(panel);
```

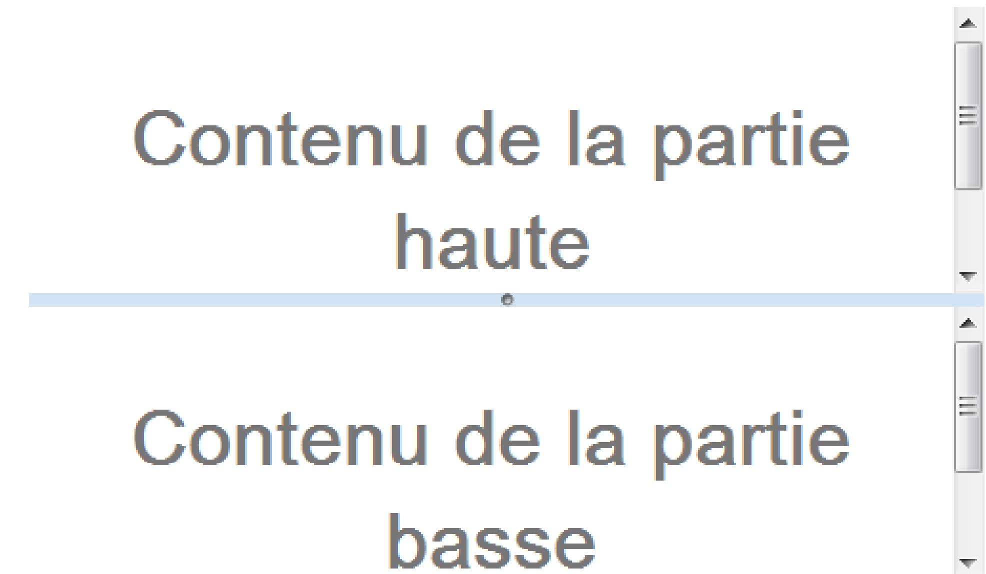


65.8.26. La classe VerticalSplitPanel

La classe VerticalSplitPanel encapsule un panneau composé de deux cellules l'une au dessus de l'autre. La taille des cellules est adaptable l'une au détriment de l'autre. Une barre de défilement est affichée pour une cellule si sa taille est trop petite pour afficher son contenu.

Exemple :

```
VerticalSplitPanel panel = new VerticalSplitPanel();
 panel.setSize("500px", "300px");
 panel.setTopWidget(new HTML("<H1>Contenu de la partie haute</H1>"));
 panel.setBottomWidget(new HTML("<H1>Contenu de la partie basse</H1>"));
 RootPanel.get("app").add(panel);
```



65.8.27. La classe HTMLTable

Cette classe abstraite est la classe mère des classes Grid et Flextable.

Les classes HTMLTable.CellFormatter, HTMLTable.ColumnFormatter et HTMLTable.RowFormatter permettent de formater respectivement le contenu d'une cellule, d'une colonne ou d'une ligne d'une table.

65.8.28. La classe LazyPanel

La classe LazyPanel est un composant qui permet de retarder son instanciation au moment de son affichage.

Ceci peut permettre d'améliorer le temps de démarrage d'une application car les parties non affichées implémentées avec un LazyPanel ne sont plus instanciées au lancement de l'application mais uniquement au moment de leur affichage.

Pour utiliser un tel panneau, il faut hériter de la classe LazyPanel et redéfinir la méthode abstraite createWidget() en incluant le code de la création du rendu du panneau.

La méthode createWidget() sera invoquéee lorsque la méthode setVisible() du panneau sera utilisée.

Exemple :

```
private static class TestLazyPanel extends LazyPanel {
  @Override
  protected Widget createWidget() {
    return new Label("Bonjour");
  }
}

public void onModuleLoad() {
  final Panel panel = new TestLazyPanel();

  panel.setVisible(false);
  PushButton bouton = new PushButton("Afficher");
  bouton.addClickHandler(new ClickHandler() {
    public void onClick(ClickEvent event) {
      panel.setVisible(true);
    }
  });

  RootPanel root = RootPanel.get("app");
  root.add(bouton);
```

```
    root.add(panel);
  }
```

65.9. La création d'éléments réutilisables

GWT permet de créer ses propres composants graphiques et permet aussi de créer des modules qui peuvent être utilisés par plusieurs projets.

65.9.1. La création de composants personnalisés

Le plus simple est de créer une classe qui hérite de la classe Composite mais il est aussi possible d'hériter d'un composant existant pour l'enrichir.

Le composant Composite permet de créer un nouveau composant par assemblage d'autres composants qui seront alors utilisables comme un seul. Il faut obligatoirement faire un appel à la méthode initWidget() à la fin du constructeur en lui passant en paramètre le panneau qui contient les éléments graphiques du composant ou un composant.

La suite de cette section sera développée dans une version future de ce document

65.9.2. La création de modules réutilisables

Il est possible de développer un module qui va contenir des composants graphiques personnalisés, des classes dédiées ou des ressources comme des images dans un module afin de permettre les réutilisation dans plusieurs applications GWT.

Ce module doit être packagé sous la forme d'un fichier .jar

La suite de cette section sera développée dans une version future de ce document

65.10. Les événements

Une application GWT est pilotée par des événements émis selon les actions de l'utilisateur sur les composants de l'application. La plupart des composants graphiques proposent l'émission d'événements en réaction aux actions de l'utilisateur sur eux.

La gestion des événements met en oeuvre des listeners d'une façon similaire à AWT ou Swing. Un listener est une interface qui doit être implémentée pour que les méthodes appelées selon l'événement contiennent les traitements à réaliser.

Des classes de types Adapter sont proposées afin de faciliter l'écriture de certains listeners : elles implémentent une interface de type Listener en définissant toutes les méthodes sans traitement. Il suffit de redéfinir la ou les méthodes requises en fonction des besoins.

Nom	Adapter	Méthodes
ChangeListener		void onChange(Widget sender)
ClickListener		void onClick(Widget sender)
EventListener		
FocusListener	FocusListenerAdapter	void onFocus(Widget sender) void onLostFocus(Widget sender)
KeyboardListener	KeyboardListenerAdapter	void onKeyDown(Widget sender, char keyCode, int modifiers) void onKeyPress(Widget sender, char keyCode, int modifiers) void onKeyUp(Widget sender, char keyCode, int modifiers)
LoadListener		void onError(Widget sender) void onLoad(Widget sender)
MouseListener	MouseListenerAdapter	void onMouseDown(Widget sender, int x, int y) void onMouseEnter(Widget sender) void onMouseLeave(Widget sender) void onMouseMove(Widget sender, int x, int y) void onMouseUp(Widget sender, int x, inty)
PopupListener		void onPopupClosed(PopupPanel sender, boolean autoClosed)
ScrollListener		void onScroll(Widget sender, int scrollLeft, int scrollTop)
TableListener		void onCellClicked(SourcedTableEvents sender, int row, int cell)
TabListener		void onBeforeTabSelected(SourcesTabEvents sender, int tabIndex) void onTabSelected(SourcesTabEvents sender, int tabIndex)
TreeListener		void onTreeItemSelected(TreeItem item) void onTreeItemStateChanged(TreeItem item)

Exemple :

```
Button b = new Button("Valider");
b.addClickListener(new ClickListener() {
  public void onClick(Widget sender) {
    // traitements lors du clic sur le bouton
  }
});
```

Exemple :

```
TextBox t = new TextBox();
t.addKeyboardListener(new KeyboardListenerAdapter() {
  public void onKeyPress(Widget sender, char keyCode, int modifiers) {
  // traitements lors de l'appui sur une touche
  }
});
```

65.11. JSNI

JSNI (JavaScript Native Interface) permet d'inclure du code Javascript dans le code Java. Cette API a plusieurs utilités :

- appel à du code Javascript non généré par GWT
- utiliser des bibliothèques de code Javascript existantes

JSNI est utilisé par le compilateur pour fusionner le code Javascript qu'il contient avec le code Javascript généré à la compilation.

Le code Javascript est inclus dans une méthode qualifiée avec le modificateur natif. Le code Javascript lui-même est inclus entre les caractères /*-{ et }-*/;

Cette séquence de caractères à l'avantage d'être ignorée par le compilateur Java et exploitée par le compilateur de GWT.

Exemple :

```
public static native void alert(String msg) /*-{
    $wnd.alert(msg);
}-*/;
```

Il est possible de passer des paramètres qui seront utilisés par le code Javascript.

Exemple :

```
public native int ajouter (int val1, int val2)
/*-{
var result = val1 + val2;
return result;
}-*/;
```

Il est possible de fournir en paramètre d'une méthode native des objets Java. Une syntaxe particulière permet d'utiliser ces objets dans le code Javascript de la méthode : soit pour accéder à un champ ou invoquer une méthode.

Pour accéder à un champ d'un objet Java dans du code Javscript, il faut utiliser la syntaxe :

objet.@classe::champ

objet est la référence sur l'objet passé en paramètre
classe est le nom pleinement qualifié de la classe de l'objet
champ est le nom du champ à accéder

Pour utiliser une méthode d'un objet Java dans du code Javascript, il faut utiliser la syntaxe :
objet.@classe::methode(signature)(parametres)

objet est la référence sur l'objet passé en paramètre
classe est le nom pleinement qualifié de la classe de l'objet
méthode est le nom de la méthode à utiliser
signature est la signature de la méthode
paramètres est la liste des paramètres si nécessaire

Il est nécessaire de préciser la signature de la méthode car celle-ci peut être surchargée et cela permet ainsi de préciser celle qui doit être utilisée. La signature est précisée en suivant une convention spéciale pour chaque type utilisable.

Type dans la signature	Type Java
B	byte
S	short
I	int
J	long
F	float
D	double
C	char
Z	boolean
Lnom/pleinement/qualifie/classe;	nom.pleinement.qualifie.classe

[type	type[]

Certaines variables spécifiques sont définies dans JSNI.

variable	Rôle
$wnd	objet Javascript Window
$doc	objet Javascript Document

Exemple :

```
public class Alert {
  public static native void alert(String msg) /*-{
    $wnd.alert(msg);
  }-*/;
}

button1.addClickListener(new ClickListener() {
public void onClick(Widget sender) {
Alert.alert("clicked!" );
}
});
```

L'inconvénient de JNSI est que le code Javascript n'est vérifiable qu'à l'exécution.

65.12. La configuration et l'internationalisation

GWT propose deux mécanismes pour internationaliser une application :

- étendre l'interface Messages pour inclure des fichiers de propriétés dans le code Javascript généré à la compilation
- utiliser la classe Dictionary pour obtenir du texte contenant les traductions

Gwt propose deux mécanismes pour faciliter la mise en oeuvre de fonctionnalités de configuration de l'application :

- configuration statique : les données de configuration sont incluses à la compilation
- configuration dynamique : les données de configuration sont incluses à l'exécution de l'application

La configuration statique est mise en oeuvre grâce aux interfaces Constants ou Messages

Il faut étendre l'une ou l'autre de ces interfaces et définir des méthodes de type getter pour chaque propriété.

La configuration dynamique est mise en oeuvre grâce à la classe Dictionary.

65.12.1. La configuration

La suite de cette section sera développée dans une version future de ce document

65.12.2. L'internationalisation

L'internationalisation (I18N) permet de fournir le support de plusieurs langues pour une application. Même si le support de plusieurs langues n'est pas prévu, il peut être intéressant d'utiliser le mécanisme d'internationalisation pour centraliser les textes affichés par l'application. Ceci permet notamment de faciliter la vérification orthographique et grammaticale et la modification des textes sans modifier le code source.

Il faut définir un fichier de propriétés stocké dans le package client. Ce fichier contient sur chaque ligne une paire clé/valeur séparée par un caractère « = ».

Exemple : le fichier MonAppMessages.properties

```
menu1=Fichier
menu2=Editer
```

Il est possible de définir des paramètres dans les valeurs. Chacun de ces paramètres est numéroté à partir de 0. Un paramètre est défini en utilisant son numéro entouré par des accolades.

Exemple :

```
erreur=La valeur saisie doit être comprise entre {0} et {1}
```

Il faut définir un fichier de propriété pour chaque langue proposée par l'application. Le nom de fichier doit être identique au fichier de propriétés initial suivi par un caractère underscore et le code langue. Les clés doivent être identiques et les valeurs doivent contenir leur traduction.

Exemple : le fichier MonAppMessages_en.properties

```
menu1=File
menu2=Edit
```

Il faut créer une interface qui porte le nom du fichier de propriétés et qui hérite de l'interface com.google.gwt.i18n.client.Messages. Il faut définir une méthode qui renvoie une chaîne de caractères pour chaque clé définie dans le fichier de propriétés. Le nom de ces méthodes doit correspondre exactement au nom de chaque clé.

Exemple :

```
package com.jmdoudoux.testgwt.client;

import com.google.gwt.i18n.client.Messages;

public interface MonAppMessages extends Messages {
  String menu1();
  String menu2();
}
```

Si des paramètres sont définis dans la valeur, il faut ajouter autant de paramètres à la méthode correspondante.

Il faut modifier le fichier de configuration de l'application. Il faut ajouter un tag <inherits name="com.google.gwt.i18n.I18N"/> pour indiquer à GWT d'ajouter le support de l'internationalisation.

Il faut aussi ajouter un tag <extend-property> possédant un attribut name dont la valeur doit être égale à la locale et un attribut values sont la valeur doit contenir le ou les codes langue supportés par l'application.

Exemple :

```
<module>
  <!-- Inherit the core Web Toolkit stuff. -->
  <inherits name='com.google.gwt.user.User' />
  <inherits name="com.google.gwt.i18n.I18N"/>
```

```
  <stylesheet src="MonApp.css" />
  <!-- Specify the app entry point class. -->
  <entry-point class='com.jmdoudoux.testgwt.client.MonApp' />

  <extend-property name="locale" values="en"/>
</module>
```

Dans le code de l'application, il faut utiliser la méthode GWT.Create() en fournissant en paramètre la classe correspondant à l'interface définie.

Exemple :

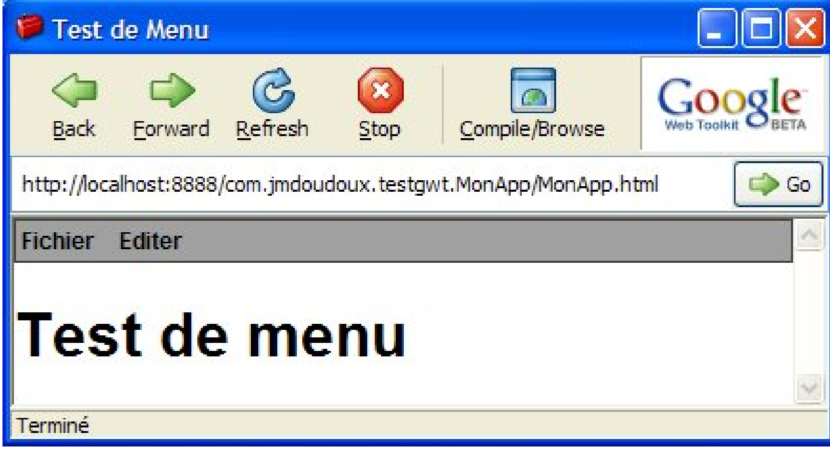
```
MonAppMessages messages = (MonAppMessages) GWT.create(MonAppMessages.class);
```

Il suffit d'utiliser l'objet instancié pour obtenir la valeur dont la clé correspond au nom de la méthode invoquée.

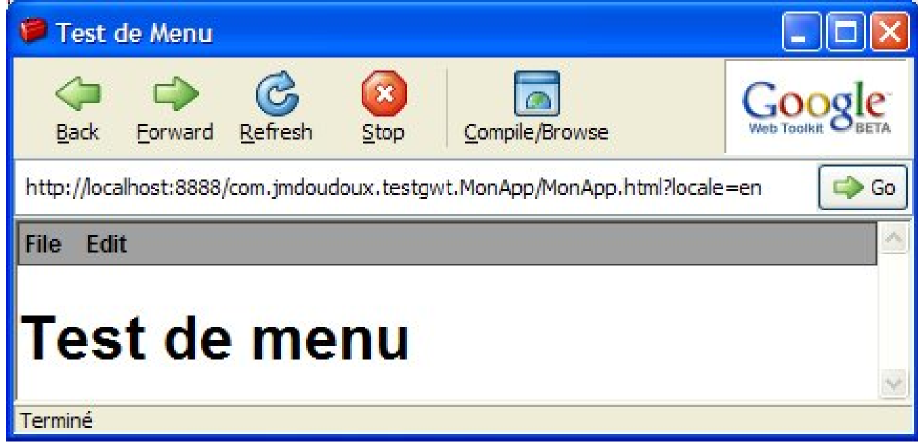
Exemple :

```
    // menu.addItem("menu1", menu1);
    menu.addItem(messages.menu1(), menu1);
    // menu.addItem("menu2", menu2);
    menu.addItem(messages.menu2(), menu2);<
```

Au lancement de l'application, la langue par défaut est utilisée.



Pour afficher l'application dans une autre langue, il faut ajouter dans l'url le paramètre locale avec comme valeur le code langue désiré.



65.13. L'appel de procédures distantes (Remote Procedure Call)

GWT propose plusieurs solutions pour permettre l'appel de traitements côté serveur :

- soumission de requêtes http
- utilisation du composant XmlHttpRequest
- utilisation du mode RPC de GWT

GWT propose des fonctionnalités pour permettre l'appel de procédures sur le serveur et ainsi mettre en oeuvre des fonctionnalités de type AJAX.

Le code côté serveur peut être réalisé avec n'importe quel langage proposant un support du traitement des requêtes HTTP. Ceci inclus Java EE notamment en utilisant des servlets. Une solution reposant sur Java côté serveur est cependant la plus facile à mettre en oeuvre.

En utilisant Java, GWT fourni deux classes qui encapsulent l'utilisation de l'objet Javascript XMLHttpRequest :

- RequestBuilder : cette classe permet d'effectuer une requête sur le serveur et d'obtenir une réponse
- GWT-RPC : c'est un mécanisme dédié de GWT qui permet l'échange d'objets Java entre le client et le serveur en utilisant un format propre à GWT

Comme dans toute application de type Ajax , il est important d'indiquer à l'utilisateur que des traitements sont en cours : cela peut se faire par exemple à l'aide d'une zone de texte spéciale, d'une image animée, d'un changement de la forme du curseur, ...

65.13.1. GWT-RPC

GWT permet aux applications de communiquer avec le serveur au travers de son propre mécanisme d'appels de type RPC. Ce mécanisme assure la sérialisation des objets qui sont échangés entre la partie cliente en Javascript et la partie serveur écrite en Java. Cette sérialisation n'est pas réalisée au travers d'un standard tel que XML, JSON, SOAP ou XML-RPC, mais elle met en oeuvre son propre format.

Les appels réalisés par l'application sont de type asynchrone.

Cette solution utilise côté serveur des servlets qui héritent de la classe RemoteServiceServlet.

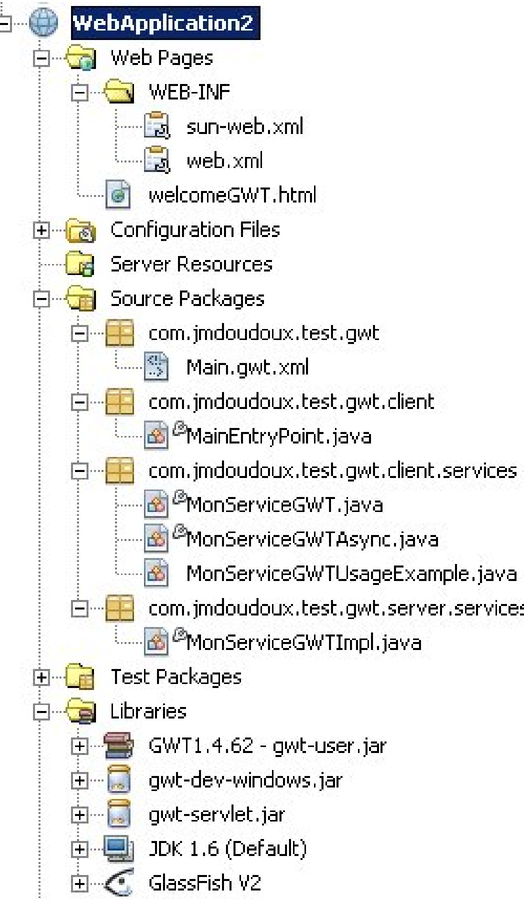
L'implémentation d'un service nécessite plusieurs étapes :

- Créer une interface qui héritent de com.google.gwt.user.client.rpc.RemoteService dans le package client de l'application
- Créer une interface pour l'appel asynchrone du service dans le package client de l'application
- Créer une servlet qui hérite de com.google.gwt.server.rpc.RemoteServiceServlet et qui implémente l'interface du service
- Déclarer la servlet dans le fichier web.xml de la webapp

65.13.1.1. Une mise oeuvre avec un exemple simple

Cette section va développer une petite application qui demande à l'utilisateur de saisir son prénom, invoque un service sur le serveur et affiche le message de salutation retourné par le service.

Exemple de projet dans Netbeans :



Dans l'exemple, les classes et interfaces sont regroupées dans un sous package services au niveau de partie client et de la partie serveur. Ceci n'est pas une obligation mais permet un meilleur découpage des sources.

GWT propose un mécanisme qui permet l'échange d'objets Java entre le client et le serveur. Pour mettre en oeuvre ce mécanisme il est nécessaire de définir trois entités :

Entités	localisation	Rôle
interface du service	Client et serveur	Décrit le service : la signature des méthodes
classe du service	Serveur	Implémentation du service
interface asynchrone du service	Client	Permet l'appel au service de façon asynchrone

L'interface du service est définie dans le sous packages client/services. Elle hérite impérativement de l'interface com.google.gwt.user.client.rpc.RemoteService et va contenir les méthodes utilisables.

Exemple :

```
package com.jmdoudoux.test.gwt.client.services;

import com.google.gwt.user.client.rpc.RemoteService;

public interface MonServiceGWT extends RemoteService {
    public String saluer(String s);
}
```

L'interface pour l'appel asynchrone du service est définie dans le sous package client/services de l'application. Par convention, elle possède le même nom que l'interface du service suffixé par Async.

Elle doit définir la méthode qui permettra l'invocation asynchrone de la méthode correspondante sur le serveur. Cette méthode doit avoir les caractéristiques suivantes :

- ne doit rien retourner,
- avoir les même paramètres que la méthode correspondante définie dans l'interface du service,
- avoir un paramètre supplémentaires de type com.google.gwt.user.client.rpc.AsyncCallback,
- ne déclarer aucune exception

Exemple :

```
package com.jmdoudoux.test.gwt.client.services;

import com.google.gwt.user.client.rpc.AsyncCallback;

public interface MonServiceGWTAsync {

    public void saluer(String s, AsyncCallback asyncCallback);
}
```

Pour la partie serveur, il faut définir une servlet dans le sous package server/services qui hérite de com.google.gwt.server.rpc.RemoteServiceServlet et qui implémente l'interface du service.

Par convention, la classe de cette servlet possède le même nom que l'interface du service suffixé par Impl puisque c'est l'implémentation concrète du service

Exemple :

```
package com.jmdoudoux.test.gwt.server.services;

import com.google.gwt.user.server.rpc.RemoteServiceServlet;
import com.jmdoudoux.test.gwt.client.services.MonServiceGWT;

public class MonServiceGWTImpl extends RemoteServiceServlet implements
        MonServiceGWT {

    public String saluer(String s) {
        return "Bonjour " + s;
    }
}
```

Remarque : pour des raisons de simplicité dans l'exemple ci-dessous la servlet implémente les traitements du service. Il serait préférable de découpler la servlet qui hérite de RemoteServiceServlet et implémente l'interface du service et de définir un objet « métier » de type POJO qui implémente l'interface du service. Chaque méthode de l'interface de la servlet se charge d'invoquer la méthode correspondante de l'objet métier.

La servlet RemoteServiceServlet héritée pour l'implémentation du service propose quelques méthodes utiles.

La méthode getThreadLocalRequest() permet d'obtenir un objet de type HttpServletRequest qui encapsule la requête http.

La méthode getThreadLocalResponse() permet d'obtenir un objet de type HttpServletResponse qui encapsule la réponse http.

En résumé, voici une synthèse des entités à créer pour un service

Entités	Hérite de	Rôle
MonServiceGWT	RemoteService	Interface qui décrit le service. Utilisé côté client et serveur
MonServiceGWTAsync		Interface pour l'appel asynchrone. Son nom est composé par convention du nom de l'interface du service suffixé par Async. Contient toutes les méthodes de l'interface du service, sans valeur de retour, sans exception et avec les même paramètres plus un dernier paramètre de type AsyncCallback

		Utilisé côté client uniquement
MonServiceGWTImpl	RemoteServiceServlet	Implémentation concrète de l'interface du service Utilisé côté serveur uniquement

L'utilisation de GWT-RPC passe par l'objet JavaScript XMLHttpRequest. Les données sont donc échangées entre le client et le serveur sous un mécanisme propre à GWT : les objets doivent donc être sérialisés côté client et désérialisés côté serveur. Côté serveur, c'est la classe RemoteServiceServlet qui automatise cette tâche.

Il faut déclarer la servlet dans le fichier web.xml de la webapp.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
  http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
    <servlet>
      <servlet-name>MonServiceGWT</servlet-name>
      <servlet-class>com.jmdoudoux.test.gwt.server.services.MonServiceGWTImpl</servlet-class>
    </servlet>
    <servlet-mapping>
      <servlet-name>MonServiceGWT</servlet-name>
      <url-pattern>/com.jmdoudoux.test.gwt.Main/services/monservicegwt</url-pattern>
    </servlet-mapping>
    <session-config>
      <session-timeout>30</session-timeout>
    </session-config>
    <welcome-file-list>
      <welcome-file>welcomeGWT.html</welcome-file>
    </welcome-file-list>
</web-app>
```

Pour utiliser le serveur Tomcat embarqué avec GWT en mode hosted, il faut déclarer la servlet dans le fichier de configuration du module pour que le client puisque invoquer le service.

Pour déclarer la servlet du service dans le fichier de configuration du module, il faut utiliser un tag servlet ayant deux attributs :

- path : chemin de mapping associé à la servlet
- class : nom pleinement qualifié de la classe de la servlet

Exemple :

```
<servlet path="/services/monservicegwt"
       class="com.jmdoudoux.test.gwt.server.services/MonServiceGWTImpl"/></pre>
```

GWT va automatiquement référencer la servlet dans le conteneur Tomcat avec le chemin fourni pour permettre son invocation par le client lors de son exécution dans le mode hosted.

GWT ne propose que des échanges asynchrones avec le serveur puisqu'ils utilisent l'objet Javascript XMLHttpRequest.

L'invocation d'un service RPC dans la partie cliente de l'application nécessite plusieurs étapes :

1. obtenir une instance de l'interface d'appel asynchrone du service en invoquant la méthode create() de la classe GWT
2. caster l'instance vers le type ServiceDefTarget
3. invoquer la méthode setServiceEntryPoint() en lui passant en paramètre l'url de la servlet qui implémente le service

4. créer une instance de la classe AsyncCallBack() qui implémente les traitements à réaliser en cas de succès et d'échec de l'appel du service
5. invoquer la méthode de l'interface appel asynchrone en lui passant en paramètre les paramètres d'appel du service et l'instance de type callback

Pour obtenir une instance de l'interface d'appel asynchrone du service, il faut invoquer la méthode create() de la classe GWT et caster le retour vers le type de l'interface asynchrone : cette instance sera le proxy qui permettra l'appel du service distant.

Pour préciser l'url d'appel du service, il faut caster l'instance du service un le type ServiceDefTarget et invoquer sa méthode seterviceEntryPoint() en lui passant en paramètre l'url.

Le service doit être hébergé sur le même domaine et le port du serveur qui a fourni la page HTML au navigateur.

Le plus simple est de créer une méthode statique qui renvoie l'instance du type de l'interface asynchrone

Exemple :

```
    public
static MonServiceGWTAsync getService() {
        MonServiceGWTAsync
service = (MonServiceGWTAsync) GWT.create(MonServiceGWT.class);
        ServiceDefTarget
endpoint = (ServiceDefTarget) service;
        String
moduleRelativeURL = GWT.getModuleBaseURL() +
"services/monservicegwt";
        endpoint.setServiceEntryPoint(moduleRelativeURL);
        return
service;
    }
```

Il faut créer une instance d'un objet qui implémente l'interface com.google.gwt.client.rpc.asyncCallback. Le plus simple est de définir une classe anonyme interne. L'interface AsyncCallback définit deux méthodes

- void onFailure(Throwable e) : callback invoqué lors de l'échec de l'invocation du service
- void onSuccess(Object result) : callback invoqué lors de la réussite de l'invocation du service

Dans la méthode onSuccess(), il faut caster l'objet passer en paramètres qui contient le résultat de l'appel vers l'objet du type adéquat.

Exemple :

```
        // Instanciation d'un callback asynchrone pour traiter la réponse
        final AsyncCallback callback = new AsyncCallback() {

            public void onSuccess(Object result) {
                lblMessage.setText((String) result);
            }

            public void onFailure(Throwable caught) {
                lblMessage.setText("Echec de la communication : " + caught.getMessage());
            }
        };
```

Pour invoquer le service, il faut obtenir une instance du proxy et invoquer la méthode voulue en lui passant en paramètres ceux à fournir au service et l'instance de l'interface AsynCallback qui prend en charge le retour de l'appel.

Exemple :

```
                        getService().saluer(text.getText(), callback);
```

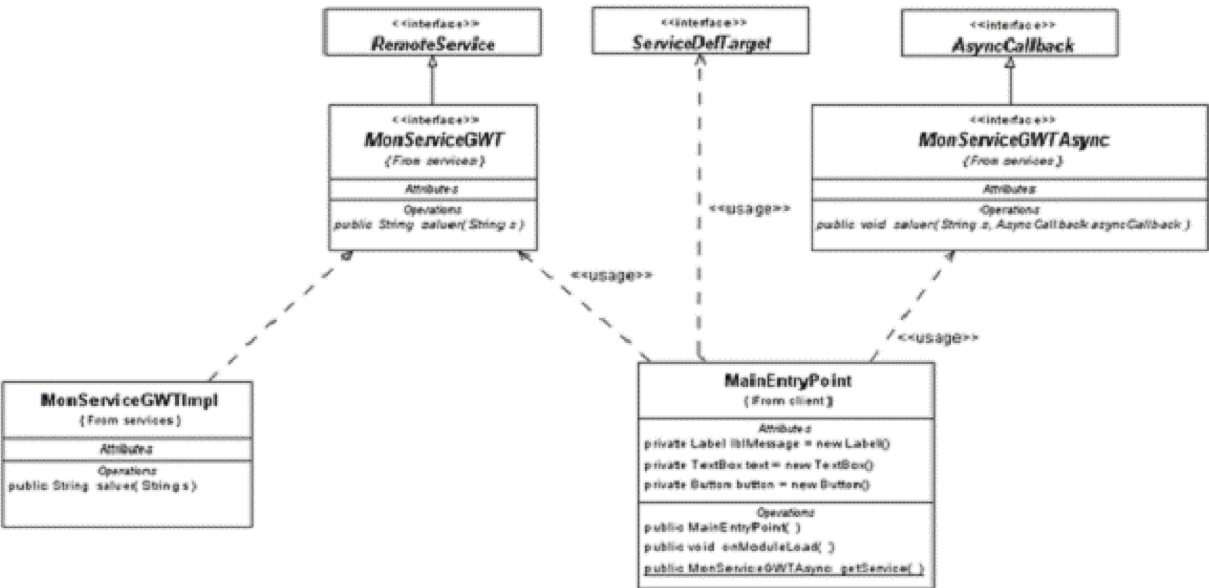
Il n'est pas possible de fournir null comme callbak même si aucun retour n'est attendu suite à l'appel au service.

L'exemple complet du code de l'application permet à l'utilisateur de saisir son prénom, d'invoquer le service et d'afficher le résultat de l'appel.

Exemple :

```
package com.jmdoudoux.test.gwt.client;

import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.core.client.GWT;
import com.google.gwt.user.client.rpc.AsyncCallback;
import com.google.gwt.user.client.rpc.ServiceDefTarget;
import com.google.gwt.user.client.ui.Button;
import com.google.gwt.user.client.ui.ClickListener;
import com.google.gwt.user.client.ui.FlowPanel;
import com.google.gwt.user.client.ui.Label;
import com.google.gwt.user.client.ui.Panel;
import com.google.gwt.user.client.ui.RootPanel;
import com.google.gwt.user.client.ui.TextBox;
import com.google.gwt.user.client.ui.Widget;
import com.jmdoudoux.test.gwt.client.services.MonServiceGWT;
import com.jmdoudoux.test.gwt.client.services.MonServiceGWTAsync;

public class MainEntryPoint implements EntryPoint {

    private Label lblMessage = new Label();
    private TextBox text = new TextBox();
    private Button button = new Button();

    public MainEntryPoint() {
    }

    public void onModuleLoad() {
        text.setText("");
        button.setText("Saluer");

        // Instanciation d'un callback asynchrone pour traiter la réponse
        final AsyncCallback callback = new AsyncCallback() {

            public void onSuccess(Object result) {
                lblMessage.setText((String) result);
            }

            public void onFailure(Throwable caught) {
                lblMessage.setText("Echec de la communication : " + caught.getMessage());
            }
        };

        button.addClickListener(new ClickListener() {

            public void onClick(Widget sender) {
                // invocation du service
                getService().saluer(text.getText(), callback);
            }
        });

        Panel main = new FlowPanel();
        RootPanel.get().add(main);
        main.add(text);
        main.add(button);
        main.add(lblMessage);
    }

    public static MonServiceGWTAsync getService() {
        MonServiceGWTAsync service = (MonServiceGWTAsync) GWT.create(MonServiceGWT.class);
        ServiceDefTarget endpoint = (ServiceDefTarget) service;
        String moduleRelativeURL = GWT.getModuleBaseURL() + "services/monservicegwt";
        endpoint.setServiceEntryPoint(moduleRelativeURL);
        return service;
    }
}
```

Le diagramme de classe ci-dessous décrit l'ensemble des classes et interfaces utilisées.



Il n'y a pas de relation au sens POO entre l'interface du service et l'interface d'appel asynchrone du service.

Pour un meilleur découpage du code source, il est possible de définir une classe dédiée qui implémente l'interface AsyncCallback

exemple : le code de l'application

Exemple :

```
package com.jmdoudoux.test.gwt.client;

import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.core.client.GWT;
import com.google.gwt.user.client.rpc.AsyncCallback;
import com.google.gwt.user.client.rpc.ServiceDefTarget;
import com.google.gwt.user.client.ui.Button;
import com.google.gwt.user.client.ui.ClickListener;
import com.google.gwt.user.client.ui.FlowPanel;
import com.google.gwt.user.client.ui.Label;
import com.google.gwt.user.client.ui.Panel;
import com.google.gwt.user.client.ui.RootPanel;
import com.google.gwt.user.client.ui.TextBox;
import com.google.gwt.user.client.ui.Widget;
import com.jmdoudoux.test.gwt.client.services.MonServiceGWT;
import com.jmdoudoux.test.gwt.client.services.MonServiceGWTAsync;

public class MainEntryPoint implements EntryPoint {

    private Label lblMessage = new Label();
    private TextBox text = new TextBox();
    private Button button = new Button();

    public MainEntryPoint() {
    }

    public void onModuleLoad() {
        text.setText("");
        button.setText("Saluer");

        button.addClickListener(new ClickListener() {

            public void onClick(Widget sender) {
                // invocation du service
```

```
                getService().saluer(text.getText(), new MonAsyncCallback(lblMessage));
            }
        });

        Panel main = new FlowPanel();
        RootPanel.get().add(main);
        main.add(text);
        main.add(button);
        main.add(lblMessage);
    }

    public static MonServiceGWTAsync getService() {
        MonServiceGWTAsync service = (MonServiceGWTAsync) GWT.create(MonServiceGWT.class);
        ServiceDefTarget endpoint = (ServiceDefTarget) service;
        String moduleRelativeURL = GWT.getModuleBaseURL() + "services/monservicegwt";
        endpoint.setServiceEntryPoint(moduleRelativeURL);
        return service;
    }
}
```

exemple : la classe MonAsyncCallback

Exemple :

```
package com.jmdoudoux.test.gwt.client;

import com.google.gwt.user.client.rpc.AsyncCallback;
import com.google.gwt.user.client.ui.Label;

public class MonAsyncCallback implements AsyncCallback {

    Label label;

    public MonAsyncCallback(Label label) {
        this.label = label;
    }

    public void onSuccess(Object result) {
        label.setStyleName("message");
        label.setText((String) result);
    }

    public void onFailure(Throwable caught) {
        label.setStyleName("erreur");
        label.setText("Echec de la communication");
    }
}
```

Il faut se souvenir dans les développements qu'une application Javascript est mono thread. Plusieurs callbacks ne peuvent donc pas être exécutés en simultané.

65.13.1.2. La transmission d'objets lors des appels aux services

Tout objet qui sera utilisé dans un échange de type GWT-RPC doit implémenter l'interface com.google.gwt.user.client.rpc.IsSerializable ou l'interface java.io.Serializable (Depuis GWT 1.4). L'usage de l'interface Serializable est recommandée car cela permet à l'objet de rester indépendant de GWT.

Les attributs déclarés transient ne seront pas sérialisés lors des échanges.

Il est nécessaire que l'objet possède un constructeur sans argument.

Exemple :

```
package com.jmdoudoux.test.gwt.client.vo;
```

```
import java.io.Serializable;

public class Personne implements Serializable {

    private String nom;
    private String prenom;
    private int taille;

    public Personne() {
    }

    public Personne(String nom, String prenom, int taille) {
        this.nom = nom;
        this.prenom = prenom;
        this.taille = taille;
    }

    public String getNom() {
        return nom;
    }

    public void setNom(String nom) {
        this.nom = nom;
    }

    public String getPrenom() {
        return prenom;
    }

    public void setPrenom(String prenom) {
        this.prenom = prenom;
    }

    public int getTaille() {
        return taille;
    }

    public void setTaille(int taille) {
        this.taille = taille;
    }
}
```

Le service peut alors utiliser l'objet en paramètre d'entrée ou de sortie.

Exemple :

```
package com.jmdoudoux.test.gwt.client.services;

import com.google.gwt.user.client.rpc.RemoteService;
import com.jmdoudoux.test.gwt.client.vo.Personne;

public interface PersonneService extends RemoteService{
    public Personne obtenirParId(int id);

    public Personne[] obtenirToutes();
}
```

L'interface d'appel asynchrone du service ne pose aucun soucis particulier puisqu'elle ne fait pas référence au bean

Exemple :

```
package com.jmdoudoux.test.gwt.client.services;

import com.google.gwt.user.client.rpc.AsyncCallback;

public interface PersonneServiceAsync {

    public abstract void obtenirParId(int id, AsyncCallback asyncCallback);
```

```
    public abstract void obtenirToutes(AsyncCallback asyncCallback);
}
```

L'implémentation du service peut utiliser toutes les API nécessaires à ces traitements notamment celles relatives aux accès à la base de données pour extraire les informations requises. Dans l'exemple ci-dessous, les données sont simplement instanciées.

Exemple :

```
package com.jmdoudoux.test.gwt.server;

import com.google.gwt.user.server.rpc.RemoteServiceServlet;
import com.jmdoudoux.test.gwt.client.services.PersonneService;
import com.jmdoudoux.test.gwt.client.vo.Personne;

public class PersonneServiceImpl extends RemoteServiceServlet implements
        PersonneService {

    public Personne obtenirParId(int id) {
        return new Personne("nom"+id,"prenom"+id,170+id);
    }

    public Personne[] obtenirToutes() {
        Personne[] resultat = new Personne[5];
        for (int i = 1 ; i < 6 ; i++) {
          resultat[i] = new Personne("nom"+i,"prenom"+i,170+i);
        }
        return resultat;
    }
}
```

Dans l'application, il suffit de caster le résultat de l'invocation du service vers le type du bean pour obtenir une instance du bean contenant les données transmises par le serveur.

Exemple :

```
package com.jmdoudoux.test.gwt.client;

import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.core.client.GWT;
import com.google.gwt.user.client.rpc.AsyncCallback;
import com.google.gwt.user.client.rpc.ServiceDefTarget;
import com.google.gwt.user.client.ui.Button;
import com.google.gwt.user.client.ui.ClickListener;
import com.google.gwt.user.client.ui.FlowPanel;
import com.google.gwt.user.client.ui.Label;
import com.google.gwt.user.client.ui.Panel;
import com.google.gwt.user.client.ui.RootPanel;
import com.google.gwt.user.client.ui.TextBox;
import com.google.gwt.user.client.ui.Widget;
import com.jmdoudoux.test.gwt.client.services.PersonneService;
import com.jmdoudoux.test.gwt.client.services.PersonneServiceAsync;
import com.jmdoudoux.test.gwt.client.vo.Personne;

public class MainEntryPoint implements EntryPoint {

    private Label lblMessage = new Label();
    private Label lblNom = new Label("Nom : ");
    private Label lblPrenom = new Label("Prénom : ");
    private Label lblTaille = new Label("Taille : ");
    private TextBox textNom = new TextBox();
    private TextBox textPrenom = new TextBox();
    private TextBox textTaille = new TextBox();
    private Button button = new Button("Obtenir données");

    public MainEntryPoint() {
    }
```

```
    public void onModuleLoad() {

        // Instanciation d'un callback asynchrone pour traiter la réponse
        final AsyncCallback callback = new AsyncCallback() {

            public void onSuccess(Object result) {
                Personne personne = (Personne) result;
                textNom.setText(personne.getNom());
                textPrenom.setText(personne.getPrenom());
                textTaille.setText(""+personne.getTaille());
            }

            public void onFailure(Throwable caught) {
                lblMessage.setStyleName("erreur");
                lblMessage.setText("Echec de la communication");
            }
        };

        button.addClickListener(new ClickListener() {

            public void onClick(Widget sender) {
                // invocation du service
                getPersonneService().obtenirParId(1, callback);
                // getService().saluer(text.getText(), new MonAsyncCallback(lblMessage));
            }
        });

        Panel main = new FlowPanel();
        RootPanel.get().add(main);
        main.add(button);
        main.add(lblNom);
        main.add(textNom);
        main.add(lblPrenom);
        main.add(textPrenom);
        main.add(lblTaille);
        main.add(textTaille);
        main.add(lblMessage);
    }

    public static PersonneServiceAsync getPersonneService() {
        PersonneServiceAsync service = (PersonneServiceAsync)
          GWT.create(PersonneService.class);
        ServiceDefTarget endpoint = (ServiceDefTarget) service;
        String moduleRelativeURL = GWT.getModuleBaseURL() + "services/personneservice";
        endpoint.setServiceEntryPoint(moduleRelativeURL);
        return service;
    }
}
```

Le résultat de l'application après un appui sur le bouton et la réception de la réponse du serveur affiche les données suivantes :

Obtenir données
Nom :
nom1
Prénom :
prenom1
Taille :
171

65.13.1.3. L'invocation périodique d'un service

Pour rafraichir périodiquement des données via un appel serveur, il faut combiner l'utilisation d'un appel RPC et d'une instance de la classe Timer.

Dans l'exemple ci-dessous, la méthode obtenirValeur() d'un service renvoie un nombre aléatoire compris entre 0 et 1000.

Exemple :

```
package com.jmdoudoux.test.gwt.server.services;

import com.google.gwt.user.server.rpc.RemoteServiceServlet;
import com.jmdoudoux.test.gwt.client.services.MonServiceGWT;

public class MonServiceGWTImpl extends RemoteServiceServlet implements
        MonServiceGWT {

    public int obtenirValeur() {
        double valeur = Math.random() * 1000;
        return (int) Math.round(valeur);
    }
}
```

Dans l'IHM de l'application, un Timer est défini : son rôle est d'invoquer toutes les secondes la méthode du service et d'afficher le résultat.

Exemple :

```
package com.jmdoudoux.test.gwt.client;

import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.core.client.GWT;
import com.google.gwt.user.client.Timer;
import com.google.gwt.user.client.rpc.AsyncCallback;
import com.google.gwt.user.client.rpc.ServiceDefTarget;
import com.google.gwt.user.client.ui.FlowPanel;
import com.google.gwt.user.client.ui.Label;
import com.google.gwt.user.client.ui.Panel;
import com.google.gwt.user.client.ui.RootPanel;
import com.jmdoudoux.test.gwt.client.services.MonServiceGWT;
import com.jmdoudoux.test.gwt.client.services.MonServiceGWTAsync;
import com.jmdoudoux.test.gwt.client.services.PersonneService;
import com.jmdoudoux.test.gwt.client.services.PersonneServiceAsync;

public class MainEntryPoint implements EntryPoint {

    private Label lblMessage = new Label();

    public MainEntryPoint() {
    }

    public void onModuleLoad() {

        Timer timer = new Timer() {

            AsyncCallback callback = new AsyncCallback() {

                public void onSuccess(Object result) {
                    lblMessage.setText(" valeur = "+ result);
                }

                public void onFailure(Throwable caught) {
                    lblMessage.setText("Echec de la communication");
                }
            };

            public void run() {
                getService().obtenirValeur(callback);
            }
        };
```

```
        timer.scheduleRepeating(1000);

        Panel main = new FlowPanel();
        RootPanel.get().add(main);
        main.add(lblMessage);
    }

    public static MonServiceGWTAsync getService() {
        MonServiceGWTAsync service = (MonServiceGWTAsync) GWT.create(MonServiceGWT.class);
        ServiceDefTarget endpoint = (ServiceDefTarget) service;
        String moduleRelativeURL = GWT.getModuleBaseURL() + "services/monservicegwt";
        endpoint.setServiceEntryPoint(moduleRelativeURL);
        return service;
    }

}
```

65.13.2. L'objet RequestBuilder

La suite de cette section sera développée dans une version future de ce document

65.13.3. JavaScript Object Notation (JSON)

La classe JSONObject encapsule un message au format JSON. Pour l'utiliser, il suffit de créer une instance de cette classe et d'utiliser la méthode put() pour ajouter une propriété en fournissant en paramètre son nom et sa valeur. La méthode toString() permet d'obtenir le message sous la forme d'une chaîne de caractères.

La suite de cette section sera développée dans une version future de ce document

65.14. La manipulation des documents XML

GWT propose un parseur XML reposant sur DOM pour permettre l'analyse ou la création de documents XML. GWT utilise le parseur du navigateur ce qui permet d'avoir de bonnes performances lors de son utilisation.

Pour utiliser les fonctionnalités de manipulation de documents XML de GWT, il faut ajouter dans la configuration du module un tag <inherits> ayant un attribut name avec la valeur « com.google.gxt.xml.XML ».

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<module>
  <inherits name="com.google.gwt.user.User"/>
  <inherits name ="com.google.gwt.xml.XML"/>
  <entry-point class="com.jmdoudoux.test.gwt.client.MainEntryPoint"/>
</module>
```

GWT propose plusieurs classes pour la mise en oeuvre de l'API DOM regroupées dans le package com.google.gwt.xml.client.

Exemple :

```
package com.jmdoudoux.test.gwt.client;

import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.user.client.Window;
import com.google.gwt.user.client.ui.Button;
import com.google.gwt.user.client.ui.ClickListener;
import com.google.gwt.user.client.ui.FlowPanel;
import com.google.gwt.user.client.ui.Label;
import com.google.gwt.user.client.ui.Panel;
import com.google.gwt.user.client.ui.RootPanel;
import com.google.gwt.user.client.ui.TextBox;
import com.google.gwt.user.client.ui.Widget;
import com.google.gwt.xml.client.Document;
import com.google.gwt.xml.client.Element;
import com.google.gwt.xml.client.Node;
import com.google.gwt.xml.client.NodeList;
import com.google.gwt.xml.client.XMLParser;

public class MainEntryPoint implements EntryPoint {

    private Label lblMessage = new Label();
    private Label lblNom = new Label("Nom : ");
    private Label lblPrenom = new Label("Prénom : ");
    private Label lblTaille = new Label("Taille : ");
    private TextBox textNom = new TextBox();
    private TextBox textPrenom = new TextBox();
    private TextBox textTaille = new TextBox();
    private Button button = new Button("Afficher données");

    public MainEntryPoint() {
    }

    public void onModuleLoad() {

        button.addClickListener(new ClickListener() {

            public void onClick(Widget sender) {
                Document doc = XMLParser.parse("<personne><nom>nom1</nom>"
                  +"<prenom>prenom1</prenom>"
                  +"<taille>170</taille></personne>");
                Element root = doc.getDocumentElement();
                NodeList children = root.getChildNodes();
                for (int i = 0 ; i < children.getLength(); i++) {
                    Node node = children.item(i);
                  Window.alert("node name="+node.getNodeName()
                    +" value="+node.getFirstChild().getNodeValue());
                }

                textNom.setText(children.item(0).getFirstChild().getNodeValue());
                textPrenom.setText(children.item(1).getFirstChild().getNodeValue());
                textTaille.setText(children.item(2).getFirstChild().getNodeValue());
            }
        });

        Panel main = new FlowPanel();
        RootPanel.get().add(main);
        main.add(button);
        main.add(lblNom);
        main.add(textNom);
        main.add(lblPrenom);
        main.add(textPrenom);
        main.add(lblTaille);
        main.add(textTaille);
        main.add(lblMessage);
    }
}
```

65.15. La gestion de l'historique sur le navigateur

Les applications utilisant AJAX modifient seulement les portions nécessaires d'une page sans la recharger entièrement : ce type d'applications est nommé SPI (Single Page Interface).

Il en résulte pour l'utilisateur une modification de ces habitudes avec le bouton Back du navigateur. Avec des applications web n'utilisant pas Ajax, l'utilisateur peut toujours revenir à la page précédente en utilisant le bouton Back.

La suite de cette section sera développée dans une version future de ce document

65.16. Les tests unitaires

Un support des tests unitaires automatisés est proposé par GWT via l'utilisation de JUnit. La version de JUnit supportée est la 3.

Pour écrire un cas de test, il faut écrire une classe qui hérite de la classe GWTTestCase. Il faut définir la méthode getModuleName() et définir les tests en écrivant la ou les méthodes commençant par test.

GWT propose un script pour générer un fichier de tests unitaires et deux scripts pour exécuter ces tests.

Exemple :

```
D:\gwt-windows-1.3.3>junitCreator
-junit D:/api/junit3.8.1/junit.jar -module com.jmdoudoux.testgwt.MonApp
-out MonAppProjet
com.jmdoudoux.testgwt.client.MonAppTests
Created
directory MonAppProjet\test\com\jmdoudoux\testgwt\client
Created file
MonAppProjet\test\com\jmdoudoux\testgwt\client\MonAppTests.java
Created
file MonAppProjet\MonAppTests-hosted.cmd
Created
file MonAppProjet\MonAppTests-web.cmd
```

La syntaxe de junitcreator est la suivante :

```
JUnitCreator -junit pathToJUnitJar -module moduleName [-eclipse projectName]
[-out dir] [-overwrite] [-ignore] className
```

Le script junitcreator possède plusieurs paramètres :

- -junit : chemin complet du fichier junit.jar (obligatoire)
- -module : nom du module GWT (obligatoire)
- -eclipse : nom du projet Eclipse dans lequel sera créé un fichier de configuration pour lancer les tests sous Eclipse
- -out : répertoire dans lequel les fichiers seront créés (par défaut le répertoire courant)
- -overwrite : remplacement des fichiers existants
- -ignore : ne pas remplacer les fichiers existants
- className : nom pleinement qualifié de la classe de tests générée

Le fichier com.jmdoudoux.testgwt.client.MonAppTests.java dans le répertoire test est créé pour servir de base aux tests.

junitCreator -junit D:/api/junit3.8.1/junit.jar -module com.jmdoudoux.testgwt -eclipse MonAppProjet -out MonAppProjet com.jmdoudoux.testgwt.client.MonAppTests

Exemple :

```
package com.jmdoudoux.testgwt.client;

import com.google.gwt.junit.client.GWTTestCase;

/**
 * GWT JUnit tests must extend GWTTestCase.
 */
public class MonAppTests extends GWTTestCase {
  /**
   * Must refer to a valid module that sources this class.
   */
  public String getModuleName() {
    return "com.jmdoudoux.testgwt.monApp";
  }
  /**
   * Add as many tests as you like.
   */
  public void testSimple() {
    assertTrue(true);
  }
}
```

Deux scripts sont créés pour exécuter les tests unitaires :

- MonAppTests-web.cmd
- MonAppTests-hosted.cmd

Pour mettre en oeuvre Junit dans un module GWT, il faut :

- ajouter le fichier junit.jar au classpath
- ajouter une entrée dans le fichier de configuration appname.gwt.xml
  <inherits name="com.google.gwt.junit.JUnit"/>
- créer une classe qui hérite de la classe com.google.gwt.junit.client.GWTTestCase
- réécrire la méthode getModuleName() pour quelle renvoie le nom pleinement qualifié du module à tester
- écrire les cas de tests sous la forme de méthode

Dans une méthode de test, il est possible de :

- Tester et modifier l'état d'un composant
- Simuler des événements
- Appeler des traitements côté serveur
- Créer des composants

La suite de cette section sera développée dans une version future de ce document

65.17. Le déploiement d'une application

La suite de cette section sera développée dans une version future de ce document

65.18. Des composants tiers

Les composants fournis en standard avec GWT sont assez basiques. Pour pouvoir développer une IHM avec des composants plus riche, il est nécessaire d'utiliser une des bibliothèques tierces proposées notamment par la communauté open source.

Il existe deux formes de composants tiers :

- native : les composants sont écrits en GWT
- wrapper : les composants encapsulent du code Javascript existant en utilisant JSNI

65.18.1. GWT-Dnd

GWT-Dnd est une bibliothèque qui propose un support pour le drag and drop dans les applications GWT.

65.18.2. MyGWT

MyGWT est une bibliothèque open source de composants pour GWT.

65.18.3. GWT-Ext

GWT-Ext est un wrapper de la bibliothèque Javascript Ext 2.0. Ext est une bibliothèque de composants Javascript très riche qui propose des composants graphiques évolués (grilles avec tri, pagination et filtre, treeview, ...)

Le site officiel du projet est à l'url http://www.gwt-ext.com

Une démo est consultable à l'url http://www.gwt-ext.com/demo/ : elle permet de visualiser toute la richesse de la bibliothèque et propose pour chaque exemple de visualiser le code source correspondant.

Produit	Version utilisée
GWT	1.4.60
GWT-Ext	2.0.1
ext	2.1

65.18.3.1. Installation et configuration

Il faut télécharger la bibliothèque GWT-Ext et décompresser le contenu de l'archive dans un répertoire du système.

Il faut ajouter le fichier gwtext.jar au classpath du projet GWT.

Il faut télécharger la bibliothèque javascript ext. La version 2.0.2 utilisée est diffusée sous licence LGPL.

Il faut décompresser l'archive téléchargée dans un répertoire du système.

Il faut créer un sous répertoire js/ext dans le sous répertoire public du projet GWT.

Il faut copier les ressources suivantes dans le sous répertoire ext créé précédemment :

* Les répertoires : adapter et resources
* Les fichiers : ext-all.js, ext-all-debug.js, ext-core.js et ext-core-debug.js

Il faut ajouter la bibliothèque dans la configuration du module :

* Il faut ajouter un tag inherits avec l'attribut name ayant pour valeur com.gwtext.GwtExt
* Il faut ajouter deux tags script avec l'attribut src ayant pour valeur js/ext/adapter/ext/ext-base.js et js/ext/ext-all.js
* Il faut ajouter un tag stylesheet avec l'attribut src ayant pour valeur /ext/resources/css/ext-all.css

Exemple :

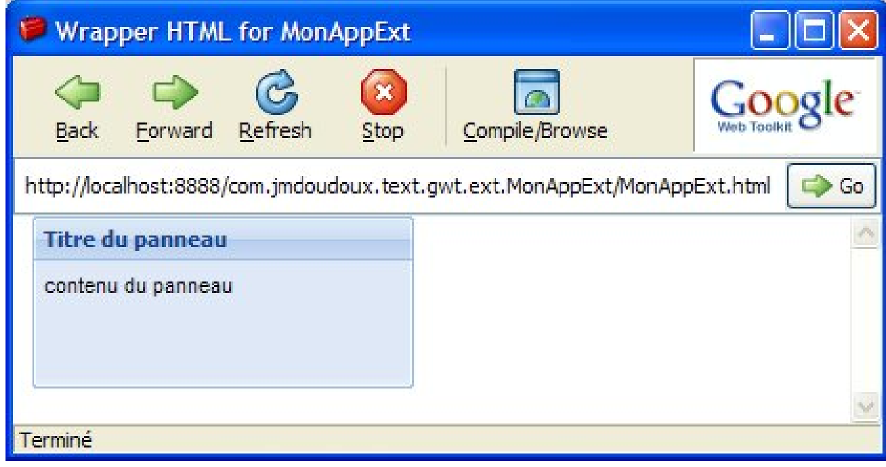
```
<module>

        <!-- Inherit the core Web Toolkit stuff.                  -->
        <inherits name='com.google.gwt.user.User'/>
      <inherits name='com.gwtext.GwtExt' />

        <!-- Specify the app entry point class.                   -->
        <entry-point class='com.jmdoudoux.text.gwt.ext.client.MonAppExt'/>

        <stylesheet src="js/ext/resources/css/ext-all.css" />
        <script src="js/ext/adapter/ext/ext-base.js" />
        <script src="js/ext/ext-all.js" />
</module>
```

65.18.3.2. La classe Panel

La classe Panel encapsule un panneau qui possède un titre et un contenu.

Exemple :

```
package com.jmdoudoux.text.gwt.ext.client;

import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.user.client.ui.RootPanel;
import com.gwtext.client.widgets.Panel;

public class MonAppExt implements EntryPoint {

  public void onModuleLoad() {
    Panel mainPanel = new Panel() {
      {
         setTitle("Titre du panneau");
         setHeight(90);
         setWidth(200);
         setFrame(true);
         setHtml("<p>Contenu du panneau</p>");
         setStyle("margin: 10px 10px 10px 10px;");
      }
   };

   RootPanel.get().add(mainPanel);
  }
}
```



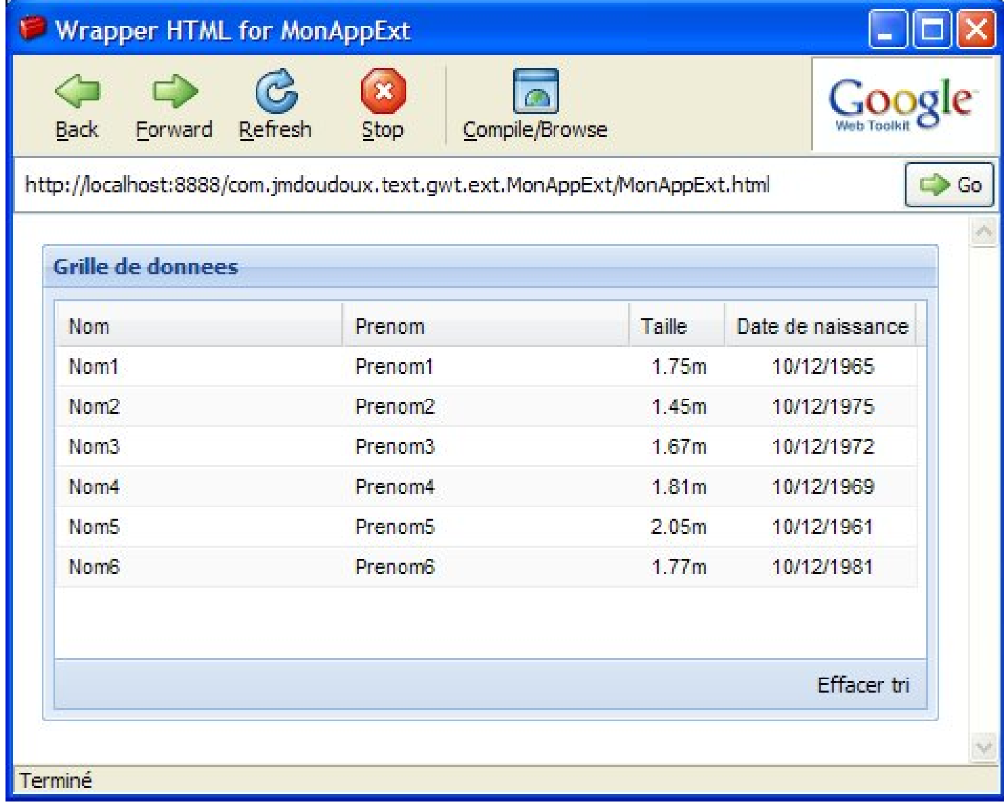
65.18.3.3. La classe GridPanel

La classe GridPanel encapsule un panneau avec un titre et une grille de données dans son contenu.

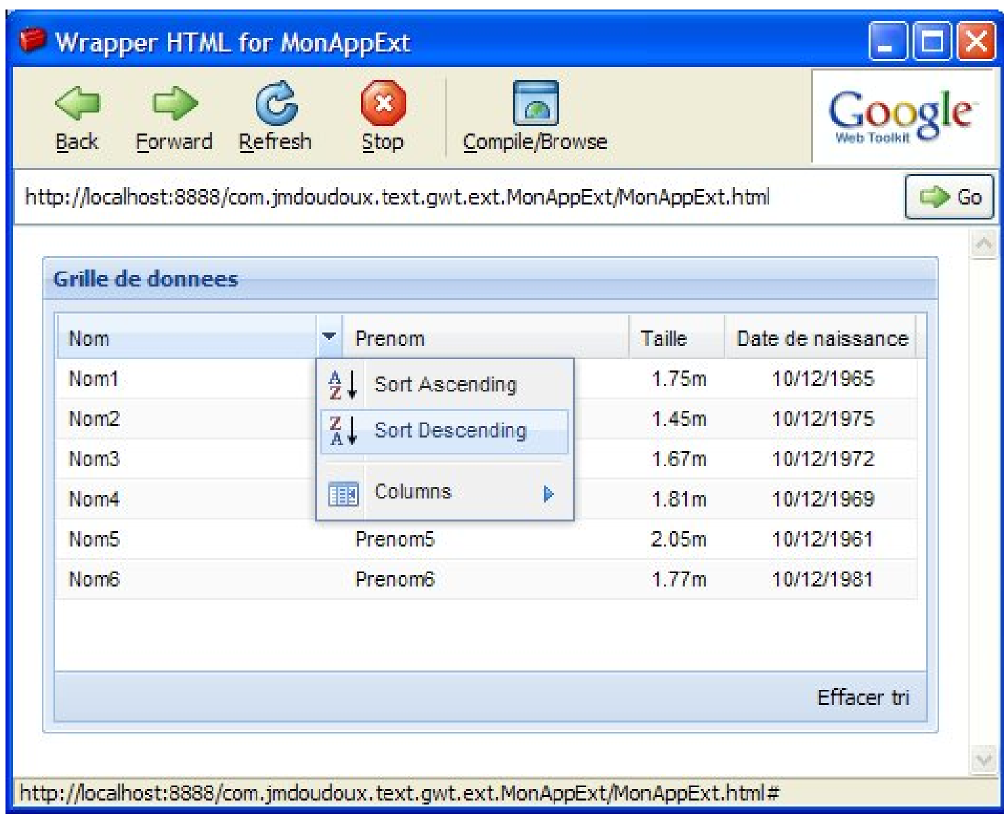
Exemple :

```
package com.jmdoudoux.text.gwt.ext.client;

import java.util.Date;

import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.i18n.client.DateTimeFormat;
import com.google.gwt.user.client.ui.RootPanel;
import com.gwtext.client.core.EventObject;
import com.gwtext.client.data.ArrayReader;
import com.gwtext.client.data.DateFieldDef;
import com.gwtext.client.data.FieldDef;
import com.gwtext.client.data.FloatFieldDef;
import com.gwtext.client.data.MemoryProxy;
import com.gwtext.client.data.Record;
import com.gwtext.client.data.RecordDef;
import com.gwtext.client.data.Store;
import com.gwtext.client.data.StringFieldDef;
import com.gwtext.client.widgets.Button;
import com.gwtext.client.widgets.Panel;
import com.gwtext.client.widgets.Toolbar;
import com.gwtext.client.widgets.ToolbarButton;
import com.gwtext.client.widgets.event.ButtonListenerAdapter;
import com.gwtext.client.widgets.grid.CellMetadata;
import com.gwtext.client.widgets.grid.ColumnConfig;
import com.gwtext.client.widgets.grid.ColumnModel;
import com.gwtext.client.widgets.grid.GridPanel;
import com.gwtext.client.widgets.grid.Renderer;

public class MonAppExt implements EntryPoint {

  private static final DateTimeFormat dateFormatter = DateTimeFormat.getFormat("M/d/y");

  public void onModuleLoad() {
    final GridPanel grille = new GridPanel();

    Panel panneau = new Panel();
    panneau.setBorder(false);
    panneau.setPaddings(15);

    RecordDef recordDef = new RecordDef(
            new FieldDef[]{
                    new StringFieldDef("nom"),
                    new StringFieldDef("prenom"),
                    new FloatFieldDef("taille"),
                    new DateFieldDef("datenais", "d/m/Y")
            }
```

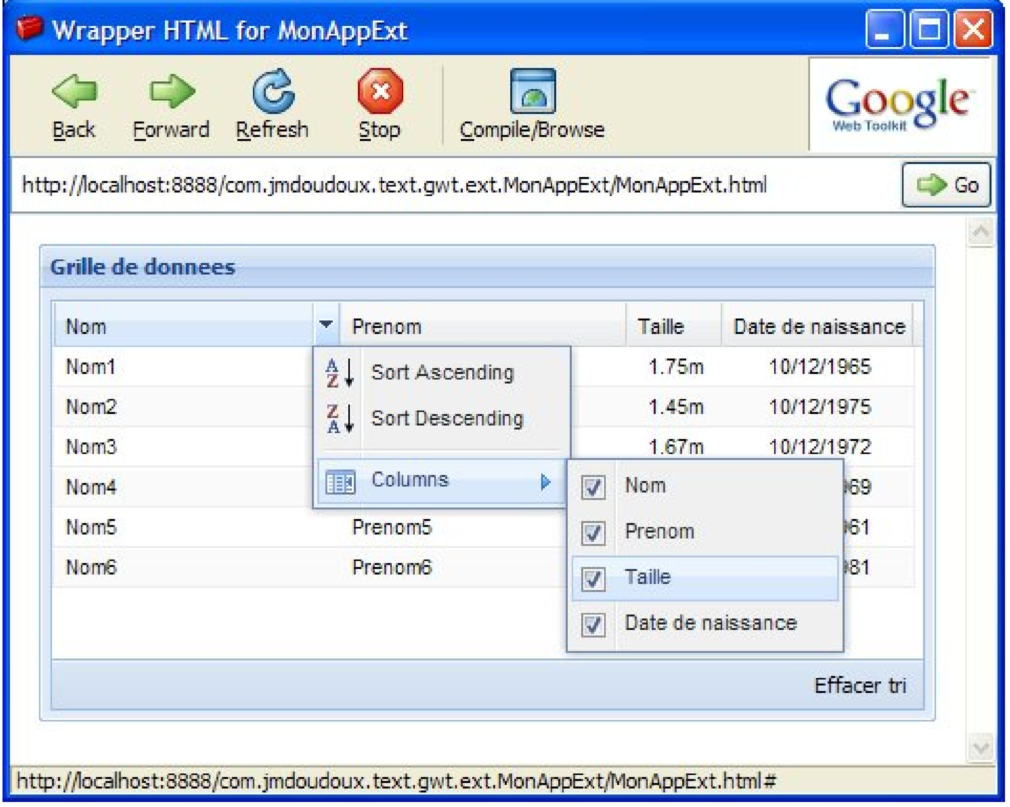
```
    );


    Object[][] donnees =  new Object[][]{
        new Object[]{"Nom1", "Prenom1",
          new Double(1.75), "13/10/1965"},
        new Object[]{"Nom2", "Prenom2",
          new Double(1.45), "13/10/1975"},
        new Object[]{"Nom3", "Prenom3",
          new Double(1.67), "13/10/1972"},
        new Object[]{"Nom4", "Prenom4",
          new Double(1.81), "13/10/1969"},
        new Object[]{"Nom5", "Prenom5",
          new Double(2.05), "13/10/1961"},
        new Object[]{"Nom6", "Prenom6",
          new Double(1.77), "13/10/1981"}
    };
    MemoryProxy proxy = new MemoryProxy(donnees);

    ArrayReader reader = new ArrayReader(recordDef);
    Store store = new Store(proxy, reader);
    store.load();
    grille.setStore(store);


    ColumnConfig[] colonnes = new ColumnConfig[]{
            new ColumnConfig("Nom", "nom",
              150, true, null, "nom"),
            new ColumnConfig("Prenom", "prenom",
              150, true, null, "prenom"),
            new ColumnConfig("Taille", "taille",
              50, true, new Renderer() {
                public String render(Object value, CellMetadata cellMetadata,
                  Record record, int rowIndex, int colNum, Store store) {
                  return "<div>" + value + "m</div>";
              }}),
            new ColumnConfig("Date de naissance", "datenais",
              100, true, new Renderer() {
                public String render(Object value, CellMetadata cellMetadata,
                  Record record, int rowIndex, int colNum, Store store) {
                  Date date = (Date)value;
                  return "<div>" +
                    dateFormatter.format(date) + "</div>";
            }})
    };

    ColumnModel columnModel = new ColumnModel(colonnes);
    grille.setColumnModel(columnModel);

    grille.setFrame(true);
    grille.setStripeRows(true);

    grille.setHeight(250);
    grille.setWidth(470);
    grille.setTitle("Grille de donnees");

    Toolbar bottomToolbar = new Toolbar();
    bottomToolbar.addFill();
    bottomToolbar.addButton(new ToolbarButton("Effacer tri",
      new ButtonListenerAdapter() {
        public void onClick(Button button, EventObject e) {
            grille.clearSortState(true);
        }
    }));
    grille.setBottomToolbar(bottomToolbar);

    panneau.add(grille);

    RootPanel.get().add(panneau);
  }
}
```



Le composant offre en standard des fonctionnalités avancées comme le tri des données d'une colonne.



Le composant permet aussi de sélectionner les colonnes qui seront affichées.



65.19. Les ressources relatives à GWT

Le site officiel de GWT : http://code.google.com/webtoolkit/

Le guide de démarrage : http://code.google.com/webtoolkit/gettingstarted.html

Le guide du développeur : http://code.google.com/webtoolkit/documentation

http://www.gwtpowered.org propose de nombreuses ressources

http://www.ongwt.com/ pour se tenir informé de l'actualité GWT

Le développement dans n'importe quel langage nécessite un ou plusieurs outils. D'ailleurs la multitude des technologies mises en oeuvre dans les projets récents nécessitent l'usage de nombreux outils.

Ce chapitre propose un recensement non exhaustif des outils utilisables pour le développement d'applications Java et une présentation détaillée de certains d'entre eux.

Le JDK fournie un ensemble d'outils pour réaliser les développements mais leurs fonctionnalités se veulent volontairement limitées au strict miminum.

Enfin, le monde open source propose de nombreux outils très utiles.

Cette partie contient les chapitres suivants :

- Les outils du J.D.K. : indique comment utiliser les outils fournis avec le JDK
- JavaDoc : explore l'outil de documentation fourni avec le JDK
- Les outils libres et commerciaux : tente une énumération non exhaustive des outils libres et commerciaux pour utiliser java
- Ant : propose une présentation et la mise en oeuvre de cet outil d'automatisation de la construction d'applications
- Maven : présente l'outil open source Maven qui facilite et automatise certaines taches de la gestion d'un projet
- Tomcat : Détaille la mise en oeuvre du conteneur web Tomcat

- Des outils open source pour faciliter le développement : présentation de quelques outils de la communauté open source permettant de simplifier le travail des développeurs.

66. Les outils du J.D.K.

Le JDK de Sun fourni un ensemble d'outils qui permettent de réaliser des applications. Ces outils sont peu ergonomiques car ils s'utilisent en ligne de commande mais en contre partie ils peuvent toujours être utilisés.

Ce chapitre contient plusieurs sections :

- Le compilateur javac
- L'interpréteur java/javaw
- L'outil jar
- L'outil appletviewer pour tester des applets
- L'outil javadoc pour générer la documentation technique
- L'outil Java Check Update pour mettre à jour Java
- La base de données Java DB
- L'outil JConsole

66.1. Le compilateur javac

Cet outil est le compilateur : il utilise un fichier source java fourni en paramètre pour créer un ou plusieurs fichiers contenant le byte code Java correspondant. Pour chaque fichier source, un fichier portant le même nom avec l'extension .class est créé si la compilation se déroule bien. Il est possible qu'un ou plusieurs autres fichiers .class soient générés lors de la compilation de la classe si celle ci contient des classes internes. Dans ce cas, le nom du fichier des classes internes est de la forme classe$classe_interne.class. Un fichier .class supplémentaire est créé pour chaque classe interne.

66.1.1. La syntaxe de javac

La syntaxe est la suivante :

javac [options] [fichiers] [@fichiers]

Cet outil est disponible depuis le JDK 1.0

La commande attend au moins un nom de fichier contenant du code source java. Il peut y en avoir plusieurs, en les précisant un par un séparé par un caractère espace ou en utilisant les jokers du système d'exploitation. Tous les fichiers précisés doivent obligatoirement posséder l'extension .java qui doit être précisée sur la ligne de commande.

Exemple : pour compiler le fichier MaClasse.

javac MaClasse.java

Exemple : pour compiler tous les fichiers sources du répertoire

javac *.java

Le nom du fichier doit correspondre au nom de la classe contenue dans le fichier source. Il est obligatoire de respecter la casse du nom de la classe même sur des systèmes qui ne sont pas sensibles à la classe comme Windows.

Depuis le JDK 1.2, il est aussi possible de fournir un ou plusieurs fichiers qui contiennent une liste des fichiers à compiler. Chacun des fichiers à compiler doit être sur une ligne distincte. Sur la ligne de commande, les fichiers qui contiennent une liste doivent être précédés d'un caractère @

Exemple :

javac @liste

Contenu du fichier liste :
test1.java
test2.java

66.1.2. Les options de javac

Les principales options sont :

Option	Rôle
-classpath path	permet de préciser le chemin de recherche des classes nécessaires à la compilation
-d répertoire	les fichiers sont créés dans le répertoire indiqué. Par défaut, les fichiers sont créés dans le même répertoire que leurs sources.
-g	génère des informations débogage
-nowarn	le compilateur n'émet aucun message d'avertissement
-O	le compilateur procède à quelques optimisations. La taille du fichier généré peut augmenter. Il ne faut pas utiliser cette option avec l'option -g
-verbose	le compilateur affiche des informations sur les fichiers sources traités et les classes chargées
-deprecation	donne des informations sur les méthodes dépréciées qui sont utilisées

66.1.3. Les principales erreurs de compilation

';' expected

Chaque instruction doit se terminer par un caractère ;. Cette erreur indique généralement qu'un caractère ; est manquant.

Exemple de code :

```
public class Test {
  int i
}
```

Résultat de la compilation :

```
javac Test.java
Test.java:3:
  ';' expected
  int i
       ^
1 error
```

'(' or '[' expected

Exemple de code :

```
public class Test {
  String[] chaines =
  new String {"aa","bb","cc"};
}
```

Résultat de la compilation :

```
C:\tmp>javac Test.java
Test.java:3:
  '(' or '[' expected
  String[] chaines = new String {"aa","bb","cc"};
                                ^
1 error
```

Cannot resolve symbol

Exemple de code :

```
import java.util;
public class Test {
}
```

Résultat de la compilation :

```
C:\tmp>javac Test.java
Test.java:1:
  cannot resolve symbol
symbol  : class util
location:
  package java
import java.util;
            ^
1 error
```

'else' without 'if'

Exemple de code :

```
public class Test {
  int i = 1;
  public void traitement() {
    if (i==0)
  {
      i =1;
    else i = 0;
  }
}
```

Résultat de la compilation :

```
C:\tmp>javac Test.java
Test.java:8:
  'else' without 'if'
    else i = 0;
    ^
Test.java:11:
  '}' expected
}
 ^
2 errors
```

'}' expected

Exemple de code :

```
public class Test {
  int i = 1;
  public void traitement() {
    if (i==0) {
      i =1;
  }
}
```

Résultat de compilation :

```
C:\tmp>javac Test.java
Test.java:9: '}' expected
}
 ^
1 error
```

Variable is already defined

Une variable est définie deux fois dans la même portée.

Exemple de code :

```
public class Test {
  int i = 1;
  int i = 0;
}
```

Résultat de la compilation :

```
C:\tmp>javac Test.java
Test.java:4: i is already defined in Test
  int i = 0;
      ^
1 error
```

Class is already defined in a single-type import

Exemple de code :

```
import java.util.List;
import java.awt.List;
public class Test {
  List liste = null;
}
```

Résultat de la compilation :

```
C:\tmp>javac Test.java
Test.java:2: java.util.List is already defined in a single-type
  import
import java.awt.List;
^
1 error
```

Reference to Class is ambiguous

Exemple de code :

```
import java.util.*;
import java.awt.*;
public
  class Test {
  List liste = null;
}
```

Résultat de la compilation :

```
C:\tmp>javac Test.java
Test.java:5:
  reference to List is ambiguous, both class java.awt.List in java.aw
t and
  class java.util.List in java.util match
  List liste = null;
  ^
1 error
```

Variable might not have been initialized

Exemple de code :

```
public class Test {
  public void traitement() {
    String chaine;
    System.out.println(chaine);
  }
}
```

Résultat de la compilation :

```
C:\tmp>javac Test.java
Test.java:6:
  variable chaine might not have been initialized
    System.out.println(chaine);
                       ^
1 error
```

Class is abstract; cannot be instantiated

Il n'est pas possible d'instancier une classe abstraite.

Exemple :

```
public abstract class Test {
  public static void main(String[] argv) {
    Test test = new Test();
  }
}
```

Résultat de la compilation :

```
C:\tmp>javac Test.java
Test.java:5:
  Test is abstract; cannot be instantiated
    Test test = new Test();
                ^
1 error
```

Non-static variable cannot be referenced from a static context

Exemple :

```
public class Test {
  int i = 0;
  public static void main(String[] argv) {
    System.out.println(i);
  }
}
```

Résultat de la compilation :

```
C:\tmp>javac Test.java
Test.java:6:
  non-static variable i cannot be referenced from a
  static context
    System.out.println(i);
                       ^
1 error
```

Cannot resolve symbol

Exemple :

```
public class Test {
  public static void main(String[] argv) {
    System.out.println(i);
  }
}
```

Résultat de la compilation :

```
C:\tmp>javac Test.java
Test.java:5:
  cannot resolve symbol
symbol  : variable i
location:
  class Test
    System.out.println(i);
                       ^
1 error
```

Class Classe is public, should be declared in a file named Classe.java

Exemple :

```
public class Test {
}
public class TestBis {
}
```

Résultat de la compilation :

```
C:\tmp>javac Test.java
Test.java:6:
  class TestBis is public, should be declared in a
  file named TestBis
.java
public class TestBis {
       ^
1 error
```

'class' or 'interface' expected

Exemple :

```
public Test {
}
```

Résultat de la compilation :

```
C:\tmp>javac Test.java
Test.java:2:
  'class' or 'interface' expected
public Test {
       ^
1 error
```

Méthode is already defined in Test

Exemple :

```
public class Test {
  public int additionner(int a, int b) {
    return a+b;
  }
  public float additionner(int a, int b) {
    return (float)a+b;
  }
}
```

Résultat de la compilation :

```
C:\tmp>javac Test.java
Test.java:8: additionner(int,int) is
  already defined in Test
  public float additionner(int a, int b) {
               ^
1 error
```

Invalid method declaration; return type required

Exemple :

```
public class Test {
  public additionner(int a, int b) {
    return a+b;
  }
}
```

Résultat de la compilation :

```
C:\tmp>javac Test.java
Test.java:4:
  invalid method declaration; return type required
  public additionner(int a, int b) {
         ^
1 error
```

Possible loss of precision

Exemple :

```
public class Test {
  public byte traitement(int a) {
    byte b = a;
    return b;
  }
}
```

Résultat de la compilation :

```
C:\tmp>javac Test.java
Test.java:5:
  possible loss of precision
found   : int
required:
  byte
    byte b = a;
             ^
1 error
```

Missing method body, or declare abstract

Exemple :

```
public class Test {
  public int additionner(int a, int b); {
    return a+b;
  }
}
```

Résultat de la compilation :

```
C:\tmp>javac Test.java
Test.java:4:
  missing method body, or declare abstract
  public int additionner(int a, int b); {
             ^
Test.java:5:
  return outside method
    return a+b;
    ^
2 errors
```

Operator || cannot be applied to int,int

Exemple :

```
public class Test {
  int i;
  public void Test(int a, int b, int c , int d) {
    if ( a = b || c = d) {
      System.out.println("test");
    }
  }
}
```

Résultat de la compilation :

```
C:\tmp>javac Test.java
Test.java:6:
  incompatible types
found   : int
required: boolean
    if ( a = b || c = d) {
         ^
Test.java:6:
  operator || cannot be applied to int,int
    if ( a = b || c = d) {
               ^
2 errors
```

Incompatible types

Exemple :

```
public class Test {
  int i;
  public void Test(int a, int b) {
    if ( a = b ) {
      System.out.println("test");
    }
  }
}
```

Résultat de la compilation :

```
C:\tmp>javac Test.java
Test.java:6:
  incompatible types
found   : int
```

```
required: boolean
    if ( a = b ) {
         ^
1 error
```

Missing return statement

Exemple :

```
public class Test {
  int a;
  public int traitement(int b) {
    a = b;
  }
}
```

Résultat de la compilation :

```
C:\tmp>javac Test.java
Test.java:7:
  missing return statement
  }
  ^
1 error
```

Modifier private not allowed here

Exemple :

```
private class Test {
}
```

Résultat de la compilation :

```
C:\tmp>javac Test.java
Test.java:2:
  modifier private not allowed here
private class Test {
        ^
1 error
```

66.2. L'interpréteur java/javaw

Ces deux outils sont les interpréteurs de byte code : ils lancent le JRE, chargent les classes nécessaires et exécutent la méthode main de la classe.

java ouvre une console pour recevoir les messages de l'application alors que javaw n'en ouvre pas.

66.2.1. La syntaxe de l'outil java

java [options] classe [argument ...]
java [options] -jar fichier.jar [argument ...]
javaw [options] classe [argument ...]
javaw [options] -jar fichier.jar [argument ...]

classe être doit un fichier .class dont il ne faut pas préciser l'extension. La classe contenue dans ce fichier doit obligatoirement contenir une méthode main(). La casse du nom du fichier doit être respectée.

Cet outil est disponible depuis la version 1.0 du JDK.

Exemple:

java MaClasse

Il est possible de fournir des arguments à l'application.

66.2.2. Les options de l'outil java

Les principales options sont :

Option	Rôle
-jar archive	Permet d'exécuter une application contenue dans un fichier .jar. Depuis le JDK 1.2
-Dpropriete=valeur	Permet de définir une propriété système sous la forme propriete=valeur. propriété représente le nom de la propriété et valeur représente sa valeur. Il ne doit pas y avoir d'espace entre l'option et la définition ni dans la définition. Il faut utiliser autant d'option -D que de propriétés à définir. Depuis le JDK 1.1
-classpath chemins ou -cp chemins	permet d'indiquer les chemins de recherche des classes nécessaires à l'exécution. Chaque répertoire doit être séparé avec un point virgule. Cette option utilisée annule l'utilisation de la variable système CLASSPATH
-classic	Permet de préciser que c'est la machine virtuelle classique qui doit être utilisée. Par défaut, c'est la machine virtuelle utilisant la technologie HotSpot qui est utilisée. Depuis le JDK 1.3
-version	Affiche des informations sur l'interpréteur
-verbose ou -v	Permet d'afficher chaque classe chargée par l'interpréteur
-X	Permet de préciser des paramètres particuliers à l'interpréteur. Depuis le JDK 1.2

L'option -jar permet d'exécuter une application incluse dans une archive jar. Dans ce cas, le fichier manifest de l'archive doit préciser qu'elle est la classe qui contient la méthode main().

66.3. L'outil jar

JAR est le diminutif de Java ARchive. C'est un format de fichier qui permet de regrouper des fichiers contenant du byte-code Java (fichier .class) ou des données utilisées en temps que ressources (images, son, ...). Ce format est compatible avec le format ZIP : les fichiers contenus dans un jar sont compressés de façon indépendante du système d'exploitation.

Les jar sont utilisables depuis la version 1.1 du JDK.

66.3.1. L'intérêt du format jar

Leur utilisation est particulièrement pertinente avec les applets, les beans et même les applications. En fait, le format jar est le format de diffusion de composants java.

Les fichiers jar sont par défaut compressés ce qui est particulièrement intéressant quelque soit leurs utilisations.

Pour une applet, le browser n'effectue plus qu'une requête pour obtenir l'applet et ses ressources au lieu de plusieurs pour obtenir tous les fichiers nécessaires (fichiers .class, images, sons ...).

Un jar peut être signé ce qui permet d'assouplir et d'élargir le modèle de sécurité, notamment des applets qui ont des droits restreints par défaut.

Les beans doivent obligatoirement être diffusés sous ce format.

Les applications sous forme de jar peuvent être exécutées automatiquement.

Une archive jar contient un fichier manifest qui permet de préciser le contenu du jar et de fournir des informations sur celui ci (classe principale, type de composants, signature ...).

66.3.2. La syntaxe de l'outil jar

Le JDK fourni un outil pour créer des archives jar : jar. C'est un outil utilisable avec la ligne de commandes comme tous les outils du JDK.

La syntaxe est la suivante :

jar [option [jar [manifest [fichier

Cet outil est disponible depuis la version 1.1 du JDK.

Les options sont :

Option	Rôle
c	Création d'une nouvelle archive
t	Affiche le contenu de l'archive sur la sortie standard
x	Extraction du contenu de l'archive
u	Mise à jour ou ajout de fichiers à l'archive : à partir de java 1.2
f	Indique que le nom du fichier contenant l'archive est fourni en paramètre
m	Indique que le fichier manifest est fourni en paramètre
v	Mode verbeux pour avoir des informations complémentaires
0 (zéro)	Empêche la compression à la création
M	Empêche la création automatique du fichier manifest

Pour fournir des options à l'outil jar, il faut les saisir sans '-' et les accoler les uns aux autres. Leur ordre n'a pas d'importance.

Une restriction importante concerne l'utilisation simultanée du paramètre 'm' et 'f' qui nécessite respectivement le nom du fichier manifest et le nom du fichier archive en paramètre de la commande. L'ordre de ces deux paramètres doit être identique à l'ordre des paramètres 'm' et 'f' sinon une exception est levée lors de l'exécution de la commande

Exemple :

```
C:\jumbo\Java\xagbuilder>jar cmf test.jar manif.mf *.class
java.io.IOException: invalid header field
        at java.util.jar.Attributes.read(Attributes.java:354)
        at java.util.jar.Manifest.read(Manifest.java:161)
        at java.util.jar.Manifest.<init>(Manifest.java:56)
        at sun.tools.jar.Main.run(Main.java:125)
        at sun.tools.jar.Main.main(Main.java:904)
```

Voici quelques exemples de l'utilisation courante de l'outil jar :

* Création d'un jar avec un fichier manifest créé automatiquement contenant tout les fichiers .class du répertoire courant

  jar cf test.jar *.class
* lister le contenu d'un jar

  jar tf test.jar
* Extraire le contenu d'une archive

  jar xf test.jar

66.3.3. La création d'une archive jar

L'option 'c' permet de créer une archive jar. Par défaut, le fichier créé est envoyé sur la sortie standard sauf si l'option 'f' est utilisée. Elle précise que le nom du fichier est fourni en paramètre. Par convention, ce fichier a pour extension .jar.

Si le fichier manifest n'est pas fourni, un fichier est créé par défaut dans l'archive jar dans le répertoire META-INF sous le nom MANIFEST.MF

Exemple (code Java 1.1) : Création d'un jar avec un fichier manifest créé automatiquement contenant tout les fichiers .class du répertoire courant

```
jar cf test.jar *.class
```

Il est possible d'ajouter des fichiers contenus dans des sous répertoires du répertoire courant : dans ce cas, l'arborescence des fichiers est conservée dans l'archive.

Exemple (code Java 1.1) : Création d'un jar avec un fichier manifest fourni contenant tous les fichiers .class du répertoire courant et tous les fichiers du répertoire images

```
jar cfm test.jar manifest.mf .class images
```

Exemple (code Java 1.1) : Création d'un jar avec un fichier manifest fourni contenant tous les fichiers .class du répertoire courant et tous les fichiers .gif du répertoire images

```
jar cfm test.jar manifest.mf *.class images/*.gif
```

66.3.4. Lister le contenu d'une archive jar

L'option 't' permet de donner le contenu d'une archive jar.

Exemple (code Java 1.1) : lister le contenu d'une archive jar

```
jar tf test.jar
```

Le séparateur des chemins des fichiers est toujours un slash quelque soit la plate-forme car le format jar est indépendant de toute plate-forme. Les chemins sont toujours donnés dans un format relatif et non pas absolu : le chemin est donné par rapport au répertoire courant. Il faut en tenir compte lors d'une extraction.

Exemple :

```
C:\jumbo\bin\test\java>jar tvf test.jar
  2156 Thu Mar 30 18:10:34 CEST 2000 META-INF/MANIFEST.MF
   678 Thu Mar 23 12:30:00 CET 2000   BDD_confirm$1.class
   678 Thu Mar 23 12:30:00 CET 2000   BDD_confirm$2.class
```

```
  4635 Thu Mar 23 12:30:00 CET 2000   BDD_confirm.class
   658 Thu Mar 23 13:18:00 CET 2000   BDD_demande$1.class
   657 Thu Mar 23 13:18:00 CET 2000   BDD_demande$2.class
   662 Thu Mar 23 13:18:00 CET 2000   BDD_demande$3.class
   658 Thu Mar 23 13:18:00 CET 2000   BDD_demande$4.class
  5238 Thu Mar 23 13:18:00 CET 2000   BDD_demande.class
   649 Thu Mar 23 12:31:28 CET 2000   BDD_resultat$1.class
  4138 Thu Mar 23 12:31:28 CET 2000   BDD_resultat.class
   533 Thu Mar 23 13:38:28 CET 2000   Frame1$1.class
   569 Thu Mar 23 13:38:28 CET 2000   Frame1$2.class
   569 Thu Mar 23 13:38:28 CET 2000   Frame1$3.class
  2150 Thu Mar 23 13:38:28 CET 2000   Frame1.class
   919 Thu Mar 23 12:29:56 CET 2000   Test2.class
```

66.3.5. L'extraction du contenu d'une archive jar

L'option 'x' permet d'extraire par défaut tous les fichiers contenus dans l'archive dans le répertoire courant en respectant l'arborescence de l'archive. Pour n'extraire que certains fichiers de l'archive, il suffit de les préciser en tant que paramètres de l'outil jar en les séparant par un espace. Pour une extraction totale ou partielle de l'archive, les fichiers sont extraits en conservant la hiérarchie des répertoires qui les contiennent.

Exemple (code Java 1.1) : Extraire le contenu d'une archive

```
jar xf test.jar
```

Exemple (code Java 1.1) : Extraire les fichiers test1.class et test2.class d'une archive

```
jar xf test.jar test1.class test2.class
```

Attention : lors de l'extraction, l'outil jar écrase tous les fichiers existants sans demander de confirmation.

66.3.6. L'utilisation des archives jar

Dans une page HTML, pour utiliser une applet fournie sous forme de jar, il faut utiliser l'option archive du tag applet. Cette option attend en paramètre le fichier jar et son chemin relatif par rapport au répertoire contenant le fichier HTML.

Exemple : le fichier HTML et le fichier MonApplet.jar sont dans le même répertoire

```
<applet code=>em<MonApplet.class>/em<
        archive=">em<MonApplet.jar>/em<"
        width=>em<300>/em< height=>em<200>/em<>
</applet>
```

Avec java 1.1, l'exécution d'une application sous forme de jar se fait grâce au jre. Il faut fournir dans ce cas le nom du fichier jar et le nom de la classe principale.

Exemple :

```
jre -cp MonApplication.jar ClassePrincipale
```

Avec Java 1.2, l'exécution d'une application sous forme de jar impose de définir la classe principale (celle qui contient la méthode main) dans l'option Main-Class du fichier manifest. Avec cette condition l'option -jar de la commande java permet d'exécuter l'application.

Exemple (Java 1.2) :

```
java -jar MonApplication.jar
```

66.3.7. Le fichier manifest

Le fichier manifest contient de nombreuses informations sur l'archive et son contenu. Ce fichier est le support de toutes les fonctionnalités particulières qui peuvent être mise en oeuvre avec une archive jar.

Dans une archive jar, il ne peut y avoir qu'un seul fichier manifest nommé MANIFEST dans le répertoire META-INF de l'archive.

Le format de ce fichier est de la forme clé/valeur. Il faut mettre un ':' et un espace entre la clé et la valeur.

Exemple :

```
C:\jumbo\bin\test\java>jar xf test.jar META-INF/MANIFEST.MF
```

Cela créé un répertoire META-INF dans le répertoire courant contenant le fichier MANIFEST.MF

Exemple :

```
Manifest-Version: 1.0
Name: BDD_confirm$1.class
Digest-Algorithms: SHA MD5
SHA-Digest: ntbIs5E5YNilE4mf570JoIF9akU=
MD5-Digest: R3zH0+m9lTFq+BlQvfQdHA==
Name: BDD_confirm$2.class
Digest-Algorithms: SHA MD5
SHA-Digest: 3QEF8/zmiTAP7MHFPU5wZyg9uxc=
MD5-Digest: swBXXptrLLwPMw/bpt6F0Q==
Name: BDD_confirm.class
Digest-Algorithms: SHA MD5
SHA-Digest: pZBT/o8YeDG4q+XrHRgrB08k4HY=
MD5-Digest: VFvY4sGRfjV1ciM9C+QIdg==
```

Dans le fichier manifest créé automatiquement avec le JDK 1.1, chaque fichier possède au moins une entrée de type 'Name' et des informations les concernant.

Entre les données de deux fichiers, il y a une ligne blanche.

Dans le fichier manifest créé automatiquement avec le JDK 1.2, il n'y a plus d'entrée pour chaque fichier.

Exemple :

```
Manifest-Version: 1.0
Created-By: 1.3.0 (Sun Microsystems Inc.)
```

Le fichier manifest généré automatiquement convient parfaitement si l'archive est utilisée uniquement pour regrouper les fichiers. Pour une utilisation plus spécifique, il faut modifier ce fichier pour ajouter les informations utiles.

Par exemple, pour une application exécutable (à partir de java 1.2) il faut ajouter une clé Main-Class en lui associant le nom de la classe dans l'archive qui contient la méthode main.

66.3.8. La signature d'une archive jar

La signature d'une archive jar joue un rôle important dans les processus de sécurité de java. La signature d'une archive permet à celui qui utilise cette archive de lui donner des droits étendus une fois que la signature a été reconnue.

Avec Java 1.1 une archive signée possède tous les droits.

Avec Java 1.2 une archive signée peut se voir attribuer des droits particuliers définis un fichier policy.

66.4. L'outil appletviewer pour tester des applets

Cet outil permet de tester une applet. L'intérêt de cet outil est qu'il permet de tester une applet avec la version courante du JDK. Un navigateur classique nécessite un plug-in pour utiliser une version particulière du JRE. Cet outil est disponible depuis la version 1.0 du JDK.

En contre partie, l'appletviewer n'est pas prévu pour tester les pages HTML. Il charge une page HTML fournie en paramètre, l'analyse, charge l'applet qu'elle contient et exécute cet applet.

La syntaxe est la suivante : appletviewer [option] fichier

L'appletview recherche le tag HTML <APPLET>. A partir du JDK 1.2, il recherche aussi les tags HTML <EMBED> et <OBJECT>.

Il possède plusieurs options dont les principales sont :

Option	Rôle
-J	Permet de passer un paramètre à la JVM. Pour passer plusieurs paramètres, il faut utiliser plusieurs options -J. Depuis le JDK 1.1
-encoding	Permet de préciser le jeu de caractères de la page HTML

L'appletviewer ouvre une fenêtre qui possède un menu avec les options suivantes :

Option de menu	Rôle
Restart	Permet d'arrêter et de redémarrer l'applet
Reload	Permet d'arrêter et de recharger l'applet
Stop	Permet d'arrêter l'exécution de l'applet. Depuis le JDK 1.1
Save	Permet de sauvegarder l'applet en la sérialisant dans un fichier applet.ser. Il est nécessaire d'arrêter l'applet avant d'utiliser cet option. Depuis le JDK 1.1
Start	Permet de démarrer l'applet. Depuis le JDK 1.1
Info	Permet d'afficher les informations de l'applet dans une boite de dialogue. Ces informations sont obtenues par les méthodes getAppletInfo() et getParameterInfo() de l'applet.
Print	Permet d'imprimer l'applet. Depuis le JDK 1.1
Close	Permet de fermer la fenêtre courante
Quit	Permet de fermer toutes les fenêtres ouvertes par l'appletviewer

66.5. L'outil javadoc pour générer la documentation technique

Cet outil permet de générer une documentation à partir des données insérées dans le code source. Cet outil est disponible depuis le JDK 1.0

La syntaxe de la commande est la suivante :

javadoc [options] [nom_packages] [fichiers_source] [-sous_packages pkg1:pkg2:...] [@fichiers_arguments]

L'ordre des arguments n'a pas d'importance :

- nom_packages : un ou plusieurs nom de packages séparés par un espace. (ces packages sont recherchés en utilisant la variable -sourcepath)
- fichiers_sources : un ou plusieurs fichiers source java séparés par un espace. Il est possible de préciser le chemin de chaque fichier. Il est aussi possible d'utiliser le caractère * pour désigner 0 ou n caractères quelconque.
- sous_packages : un ou plusieurs sous packages à inclure dans la documentation
- @fichiers_arguments : un ou plusieurs fichiers qui contiennent les options à utiliser par Javadoc

Il faut fournir en paramètres de l'outil javadoc un nom de package ou un ensemble de fichier source java.

Les principales options utilisables sont :

Option	Rôle
-classpath chemin	Classpath dans lequel l'outil va rechercher les fichiers sources et .class Cette option permet de remplacer le classpath standard par celui fourni.
-encoding name	Précise le jeu de caractères utilisés dans les fichiers sources
-J flag	Permet de passer un argument à la JVM dans laquelle s'exécute l'outil javadoc Exemple : -J-Xmx32m -J-Xms32m
-sourcepath chemins	Chemins dans lequel l'outil va rechercher les fichiers sources à documenter. Plusieurs chemins peuvent être précisé en utilisant le caractère ; comme séparateur. Par défaut, si ce paramètre n'est pas précisé, c'est le classpath qui est utilisé.
-verbose	Affiche des informations sur la génération en cours
-help	Affiche un résumé des options de la commande
-doclet classe	Permet de préciser un doclet personnalisé (la classe fournie en paramètre doit être pleinement qualifiée).
-docletpath	Permet de préciser le chemin du doclet personnalisé
-exclude packages	Permet de préciser une liste de packages qui ne seront pas pris en compte par l'outil. Le caractère : est utilisé comme séparateur entre chaque packages
-package	Inclue seulement les membres et classes package friendly, protected et public
-private	Inclue tous les membres et classes
-protected	Inclue seulement les membres et classes protected et public
-public	Inclue seulement les membres et classes public
-overview fichier	Utilise le fichier précisé comme fichier overview-summary.html dans la documentation générée
-source version	Permet de préciser la version de java utilisée par le code source. Les valeurs possibles sont 1.3, 1.4 et 1.5. Par défaut, c'est la version du JDK qui est utilisée

-subpackages packages	Permet de préciser les packages qui seront pris en compte. Le caractère : est utilisé comme séparateur entre chaque packages

L'outil javadoc utiliser un doclet pour réaliser le rendu de la documentation générée : il utilise un doclet par défaut si aucun autre doclet n'est spécifié via l'option -doclet.

Les principales options du doclet standard utilisables sont :

Option	Rôle
-author	Permet de demander la prise en compte des tags @author dans la documentation générée
-bottom	Permet d'insérer du code HTML dans le pied de chaque page
-charset nom	Permet de préciser le jeu de caractère des fichiers HTML générés
-d repertoire	Précise le répertoire dans lequel la documentation va être générée. Par défaut, c'est le répertoire courant qui est utilisé
-docencoding nom	Précise le jeu de caractères utilisés dans les fichiers générés
-doctitle titre	Permet de préciser le titre de la page index (il est possible d'utiliser du code HTML : dans se cas entourer le titre avec des caractères " ...")
-footer	Permet d'insérer du code HTML à droite du footer par défaut
-help fichier	Permet d'utiliser un fichier d'aide personnalisé. Le fichier doit être au format HTML
-header	Permet d'insérer du code HTML à droite du header à défaut
-link url	Permet de générer des liens vers une documentation générée (par exemple celle du JDK). L'url peut être relative ou absolue. Exemple : ***-link http://java.sun.com/j2se/1.5.0/docs/api***
-linksource	Permet d'inclure dans la documentation le code source au format HTML avec une numérotation des lignes
-nodecprecatedlist	Permet de ne pas générer le fichier deprecated-list.html
-nodeprecated	Permet de ne pas intégrer les tags @deprecated dans la documentation
-nohelp	Permet de ne pas générer le lien "Help"
-noindex	Permet de ne pas générer la page index des packages
-nonavbar	Permet de ne pas générer la barre de navigation
-nosince	Permet de ne pas intégrer les tags @since dans la documentation
-notree	Permet de ne pas générer la hiérarchie des classes
-splitindex	Permet de générer l'index sous la forme d'un fichier par lettre
-stylesheetfile fichier	Permet de préciser la feuille de style utilisée dans la documentation
-use	Permet de demander la génération des pages d'utilisation des classes et packages
-version	Permet de demander la prise en compte des tags @version dans la documentation générée
-windowtitle titre	Permet de préciser le titre des pages

Il est possible de préciser les paramètres de l'outil javadoc (sauf l'option -J) dans un ou plusieurs fichiers en passant leur nom précédés par le caractère @. Dans un tel fichier les paramètres peuvent être séparés par un espace ou un retour chariot.

Il n'est pas possible d'utiliser le caractère joker * dans les noms de fichiers ni de faire un référence à un autre fichier de paramètre avec le caractère @.

Exemple :

C:>javadoc @javadocparam

Exemple : le fichier javadocparam
`-d documentation -use -splitindex`

Remarque : l'outil javadoc ne sait pas travailler de façon incrémentale : toute la documentation est à regénérer à chaque fois.

Des informations supplémentaires sur les éléments à inclure dans le code source sont fournies dans le chapitre «JavaDoc».

66.6. L'outil Java Check Update pour mettre à jour Java

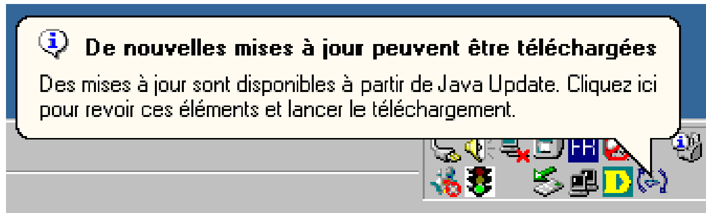
Jucheck (Java Update Check) est un outil proposé par Sun pour permettre une mise à jour automatique de l'environnement d'exécution Java.

L'outil jusched.exe est installé par défaut et configuré pour une exécution automatique depuis la version 1.4.2 du J2SE. Il permet une automatisation de l'exécution de jucheck.exe.

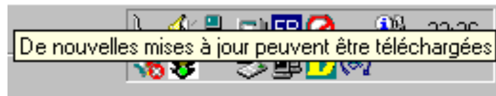
Pour lancer manuellement la mise à jour, il suffit d'exécuter le programme jucheck.exe dans le répertoire bin du JRE.

Si aucune mise à jour n'est disponible, un message est affiché :

Sinon une bulle d'aide informe que des mises à jour peuvent être téléchargées.



En laissant le curseur de la souris sur l'icône du programme de mise à jour, une bulle d'aide est affichée.

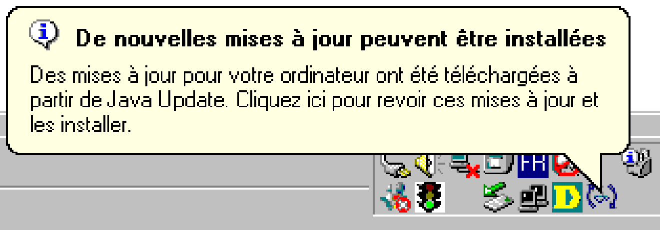


Pour télécharger les mises à jour, il suffit d'utiliser l'option " Télécharger " du menu contextuel associé à l'icône

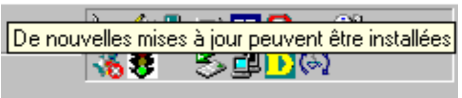
Une boîte de dialogue permet de demander le téléchargement des éléments dont la version est indiquée.

Cliquez sur le bouton " Téléchargement ".

Une fois le téléchargement terminée, une bulle est affichée.



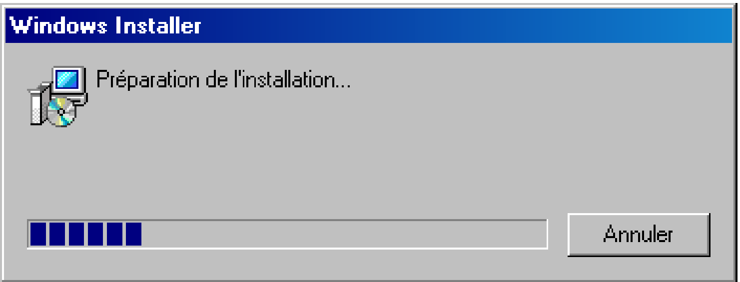
En laissant le curseur de la souris sur l'icône du programme de mise à jour, une bulle d'aide est affichée.



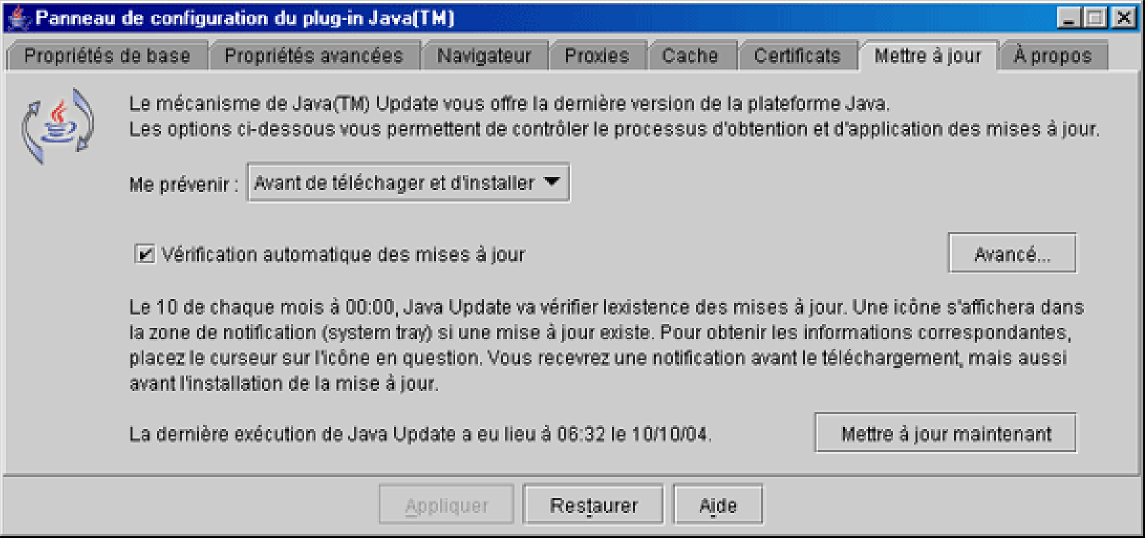
Pour installer les mises à jour, il suffit d'utiliser l'option " Installer " du menu contextuel associé à l'icône

Cliquez sur le bouton " Installer ".

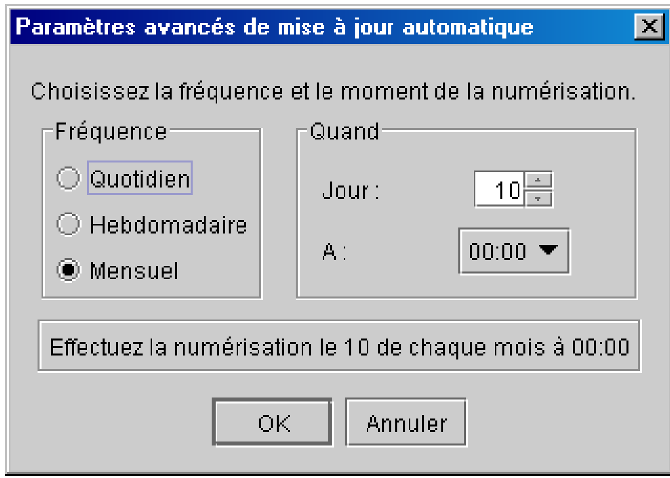
L'assistant se lance pour diriger les différentes étapes



L'option " Propriétés " permet d'ouvrir une boîte de dialogue pour gérer les paramètres des mises à jour dans la l'onglet " Mettre à jour ".



Le bouton " Avancé " permet de définir les paramètres de recherche automatique.



66.7. La base de données Java DB

Java SE 6 intègre une base de données : Java DB. C'est en fait la base de données open source Apache Derby écrite entièrement en Java. C'est une base de données légère (2 Mb) qui propose cependant des fonctionnalités intéressantes (gestion des transactions et des accès concurrents, support des triggers et des procédures stockées, ...)

Java DB est donc le nom sous lequel Sun propose la base de données open source Derby du groupe Apache dans certains de ces outils notamment le JDK depuis sa version 6.0.

Java DB est stocké dans le sous répertoire db du répertoire d'installation d'un JDK. Avec le JDK 6.0 c'est la version 10.2 de Java DB qui est fournie.

Remarque : dans cette section Java DB peut être remplacée par Derby et vice et versa.

L'ajout de Java DB dans le JDK permet de rapidement écrire des applications qui utilisent une base de données et les fonctionnalités proposées par la version 4.0 de JDBC.

Java DB est idéale dans un environnement de développement car il est riche en fonctionnalités, facile à utiliser, multi-plateforme puisqu'écrit en Java et peut être mis en oeuvre sur une simple machine de développement.

Java DB peut fonctionner selon deux modes :

- Embedded : la base de données est exécutée comme une partie de l'application
- Client/server : la base de données est exécutée de façon indépendante de l'application

Java DB peut être intégré dans une application (mode Embedded) ce qui évite d'avoir à installer et configurer une base de données lors du déploiement de l'application.

Java DB propose plusieurs outils pour assurer sa gestion.

L'outil ij permet d'exécuter des scripts sur une base de données.

Le lancement de l'outil ij peut se faire de deux façons :

Exemple :

```
C:\Program Files\Java\jdk1.6.0\db>java -jar lib\derbyrun.jar ij
version ij 10.2
ij>
```

ou

Exemple :

```
C:\Program Files\Java\jdk1.6.0\db>java -cp .\lib\derbytools.jar org.apache.derby
.tools.ij
ij version 10.2
ij>
```

La création d'une base de données nommée MaBaseDeTest et la connexion à cette nouvelle base se fait en utilisant l'outil ij.

Exemple :

```
ij> CONNECT 'jdbc:derby:MaBaseDeTest;create=true';
```

Remarque : l'outil ij peut être utilisé avec n'importe quel pilote JDBC.

Exemple : Création de la table Personne

```
ij> CREATE TABLE PERSONNE (ID INT PRIMARY KEY, NOM VARCHAR(50), PRENOM VARCHAR(5
0));
0 lignes insérés/mises à jour/supprimées

ij> select * from PERSONNE;
ID         |NOM                                               |PRENOM

--------------------------------------------------------------------------------
---------------------------------

0 lignes sélectionnées
```

Exemple : Ajout d'occurrences dans la table Personne

```
ij> INSERT INTO PERSONNE VALUES (1,'nom1','prenom1'), (2,'nom2','prenom2'), (3,'
nom3','prenom3');
3 lignes insérées/mises à jour/supprimées
ij> select * from PERSONNE;
ID         |NOM                                               |PRENOM

--------------------------------------------------------------------------------
---------------------------------
1          |nom1                                              |prenom1

2          |nom2                                              |prenom2

3          |nom3                                              |prenom3


3 lignes sélectionnées
```

Il est alors possible d'utiliser JDBC pour se connecter à la base de données et effectuer des opérations sur ces données.

La commande exit; permet de quitter l'outil ij.

Une fois la base de données créée, il est possible de l'utiliser par exemple en utilisant l'API JDBC.

Exemple : utilisation de Java DB en mode embedded

```
package com.jmdoudoux.test.jpa;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.Statement;
```

```
public class TestDerby {

  private static String dbURL =
    "jdbc:derby:C:/Program Files/Java/jdk1.6.0/db/MaBaseDeTest;user=APP";

  public static void main(String[] args) {

    try {
      Connection conn = null;
      Statement stmt = null;

      Class.forName("org.apache.derby.jdbc.EmbeddedDriver").newInstance();
      conn = DriverManager.getConnection(dbURL);

      stmt = conn.createStatement();
      ResultSet results = stmt.executeQuery("select * from personne");
      ResultSetMetaData rsmd = results.getMetaData();
      int nbColonnes = rsmd.getColumnCount();
      for (int i = 1; i <= nbColonnes; i++) {
        System.out.print(rsmd.getColumnLabel(i) + "\t\t");
      }

      System.out.println("\n-------------------------------------------------");

      while (results.next()) {
        System.out.println(results.getInt(1) + "\t\t"
          + results.getString(2) + "\t\t" + results.getString(3));
      }
      results.close();
      stmt.close();
      if (conn != null) {
        conn.close();
      }
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

}
```

Remarque : par défaut dans le mode embeded, les tables sont créées dans le schéma APP.

Pour utiliser Java DB en mode client/serveur, il suffit de remplacer le pilote JDBC par org.apache.derby.jdbc.ClientDriver et de changer dans l'url le chemin local par l'url de la base de données.

L'outil dblook permet de générer le DDL d'une base de données.

Exemple :

```
C:\Program Files\Java\jdk1.6.0\db>java -jar lib\derbyrun.jar dblook -d jdbc:derb
y:MaBaseDeTest
-- Horodatage : 2007-06-28 15:42:21.613
-- La base de données source est : MaBaseDeTest
-- L'URL de connexion est : jdbc:derby:MaBaseDeTest
-- appendLogs: false

-- ----------------------------------------------
-- Instructions DDL pour tables
-- ----------------------------------------------

CREATE TABLE "APP"."ADRESSE" ("ID_ADRESSE" INTEGER NOT NULL, "RUE" VARCHAR(250)
NOT NULL, "CODEPOSTAL" VARCHAR(7) NOT NULL, "VILLE" VARCHAR(250) NOT NULL);

CREATE TABLE "APP"."PERSONNE" ("ID" INTEGER NOT NULL, "NOM" VARCHAR(50), "PRENOM
" VARCHAR(50));

-- ----------------------------------------------
-- Instructions DDL pour clés
-- ----------------------------------------------
```

```
-- primaire/unique
ALTER TABLE "APP"."ADRESSE" ADD CONSTRAINT "SQL070628114148710" PRIMARY KEY ("ID
_ADRESSE");

ALTER TABLE "APP"."PERSONNE" ADD CONSTRAINT "SQL070627114331220" PRIMARY KEY ("I
D");
```

L'outil sysinfo permet d'obtenir des informations sur l'environnement et sur la base de données

Exemple :

```
C:\Program Files\Java\jdk1.6.0\db>java -cp .\lib\derbytools.jar org.apache.derby
.tools.ij
ij version 10.2
ij> exit;

C:\Program Files\Java\jdk1.6.0\db>java -jar lib\derbyrun.jar sysinfo
------------------ Informations Java  ------------------
Version Java :    1.6.0_01
Fournisseur Java :     Sun Microsystems Inc.
Répertoire principal Java :       C:\Program Files\Java\jre1.6.0_01
Chemin de classes Java :  lib\derbyrun.jar
Nom du système d'exploitation :         Windows XP
Architecture du système d'exploitation :         x86
Version du système d'exploitation :         5.1
Nom d'utilisateur Java :  JMD
Répertoire principal utilisateur Java :  C:\Documents and Settings\jmd
Répertoire utilisateur Java :   C:\Program Files\Java\jdk1.6.0\db
java.specification.name: Java Platform API Specification
java.specification.version: 1.6
--------- Informations Derby --------
JRE - JDBC: Java SE 6 - JDBC 4.0
[C:\Program Files\Java\jdk1.6.0\db\lib\derby.jar] 10.2.1.7 - (453926)
[C:\Program Files\Java\jdk1.6.0\db\lib\derbytools.jar] 10.2.1.7 - (453926)
[C:\Program Files\Java\jdk1.6.0\db\lib\derbynet.jar] 10.2.1.7 - (453926)
[C:\Program Files\Java\jdk1.6.0\db\lib\derbyclient.jar] 10.2.1.7 - (453926)
------------------------------------------------------
----------------- Informations sur lenvironnement local -----------------
Environnement local actuel :  [français/Luxembourg [fr_LU]]
La prise en charge de cet environnement local a été trouvée : [de_DE]
         version : 10.2.1.7 - (453926)
La prise en charge de cet environnement local a été trouvée : [es]
         version : 10.2.1.7 - (453926)
La prise en charge de cet environnement local a été trouvée : [fr]
         version : 10.2.1.7 - (453926)
La prise en charge de cet environnement local a été trouvée : [it]
         version : 10.2.1.7 - (453926)
La prise en charge de cet environnement local a été trouvée : [pt_BR]
         version : 10.2.1.7 - (453926)
------------------------------------------------------
```

Pour démarrer JavaDB en mode client-server, il faut exécuter l'application org.apache.derby.drda.NetworkServerControl

Exemple :

```
C:\Program Files\Java\jdk1.6.0\db>java -cp .\lib\derbyrun.jar org.apache.derby.d
rda.NetworkServerControl start -h localhost
Le serveur est prêt à accepter les connexions au port 1527.
```

L'option -h permet de préciser le serveur.
L'option - p permet de préciser le port à utiliser si le port 1527 utilisé par défaut ne convient pas.

66.8. L'outil JConsole

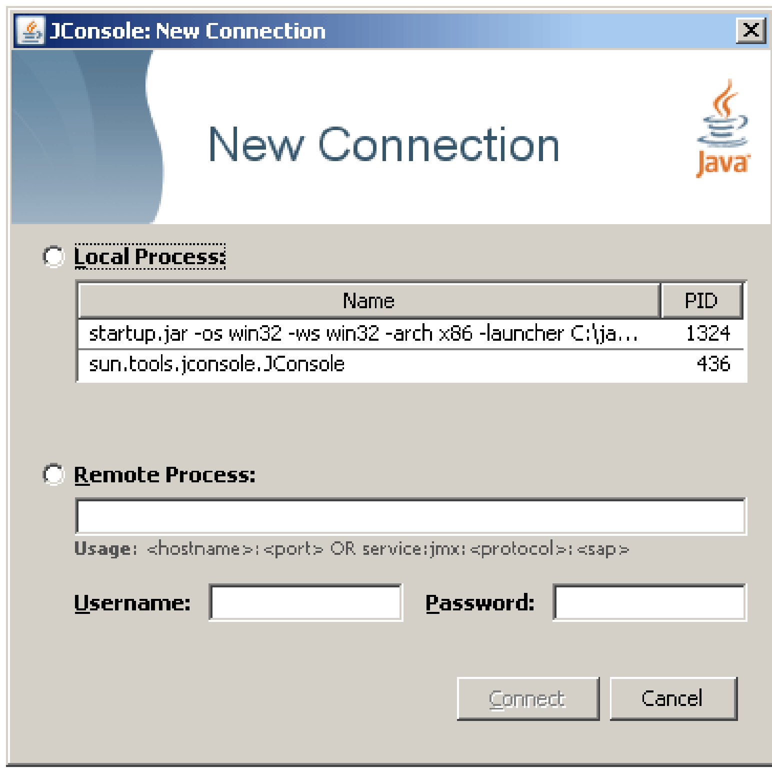
Depuis Java SE 5.0, le JDK propose l'outil JConsole qui est une interface de monitoring et de management utilisant JMX. JConsole est une application graphique qui un client JMX permettant de mettre en oeuvre la plupart des fonctionnalités de JMX.

JConsole est dans le sous répertoire bin du répertoire d'installation du JDK.

Remarque : Il n'est pas recommandé d'utiliser JConsole en production car son utilisation consomme beaucoup de ressources. Si son utilisation est nécessaire, il est recommandé d'exécuter JConsole sur un système distant de celui de la JVM à monitorer.

Pour utiliser JConsole , il suffit d'exécuter jconsole dans une nouvelle boîte de commande.

Remarque : sous Windows, JMX ne fonctionne correctement que si la partition du système est formatée en NTFS.

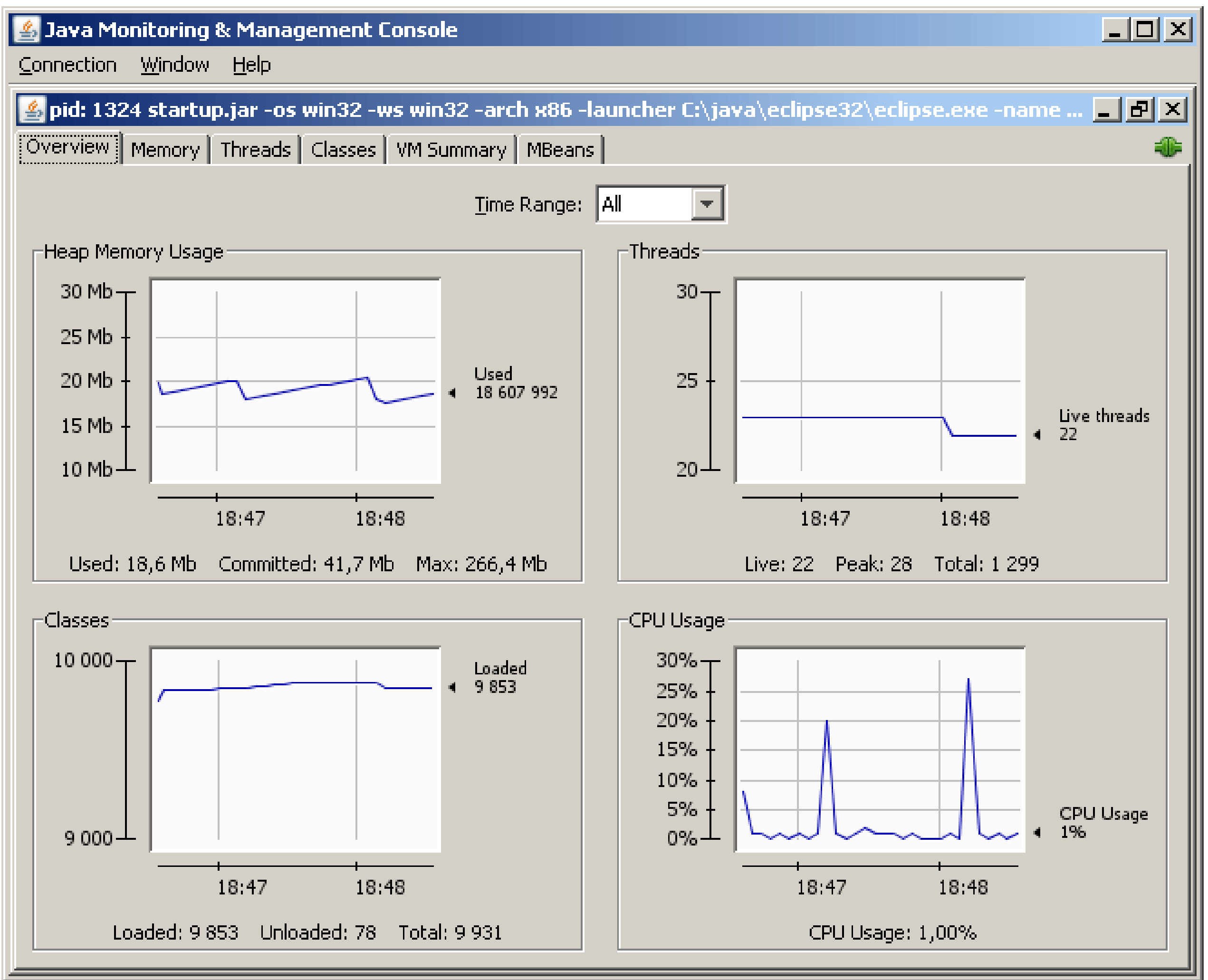


Une fenêtre de connexion permet de préciser quel sera le processus d'une JVM à utiliser.

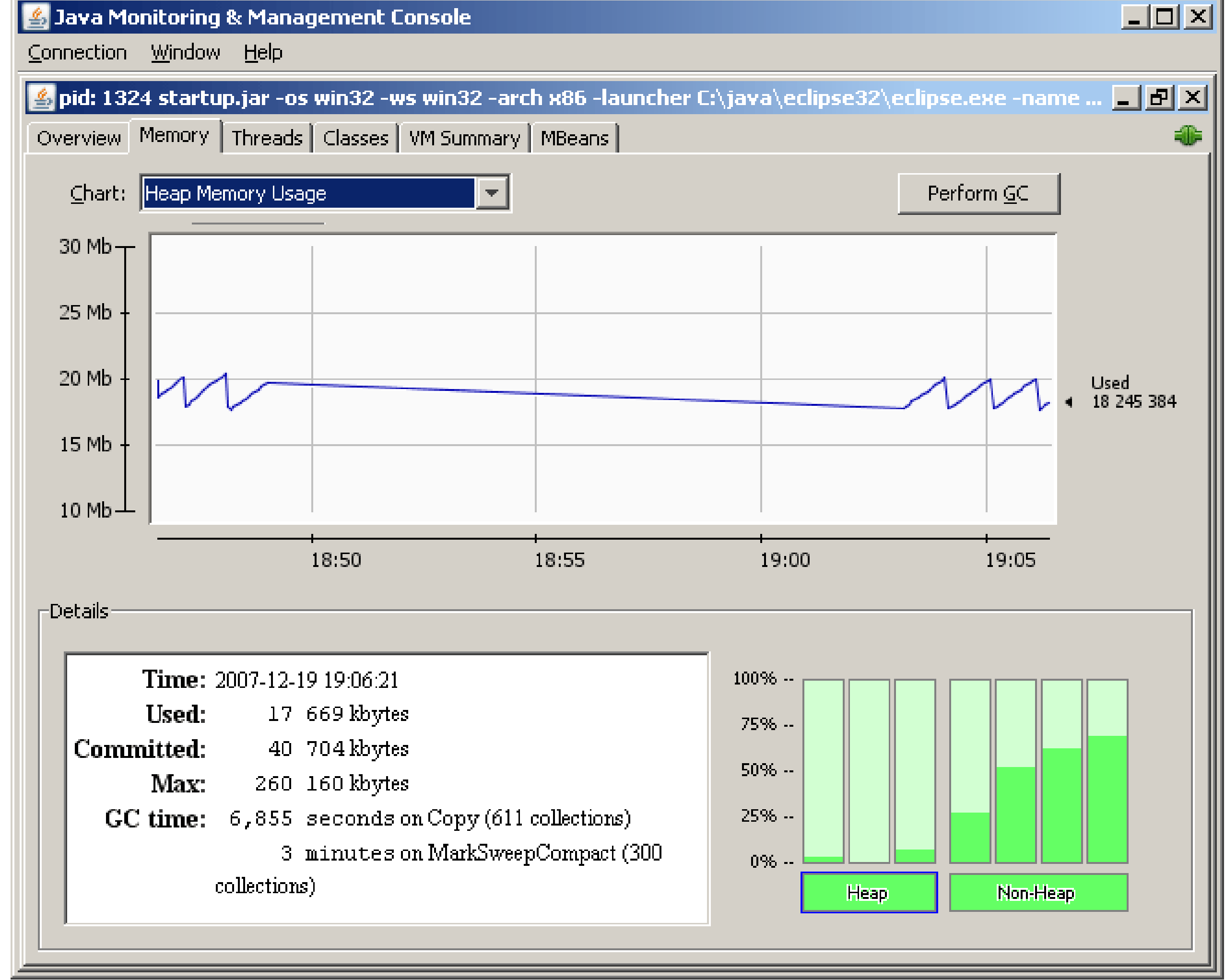
Pour une JVM locale, il suffit de sélectionner « Local Process », de sélectionner la JVM parmi celle proposée et de cliquer sur « Connect »

Pour une connexion distante, il faut saisir l'url de connexion et éventuellement le user et le mot de passe si l'authentification est activée.

Une fois la connexion établie, une fenêtre contenant plusieurs onglets permet d'avoir des informations sur l'état de différentes fonctionnalités.



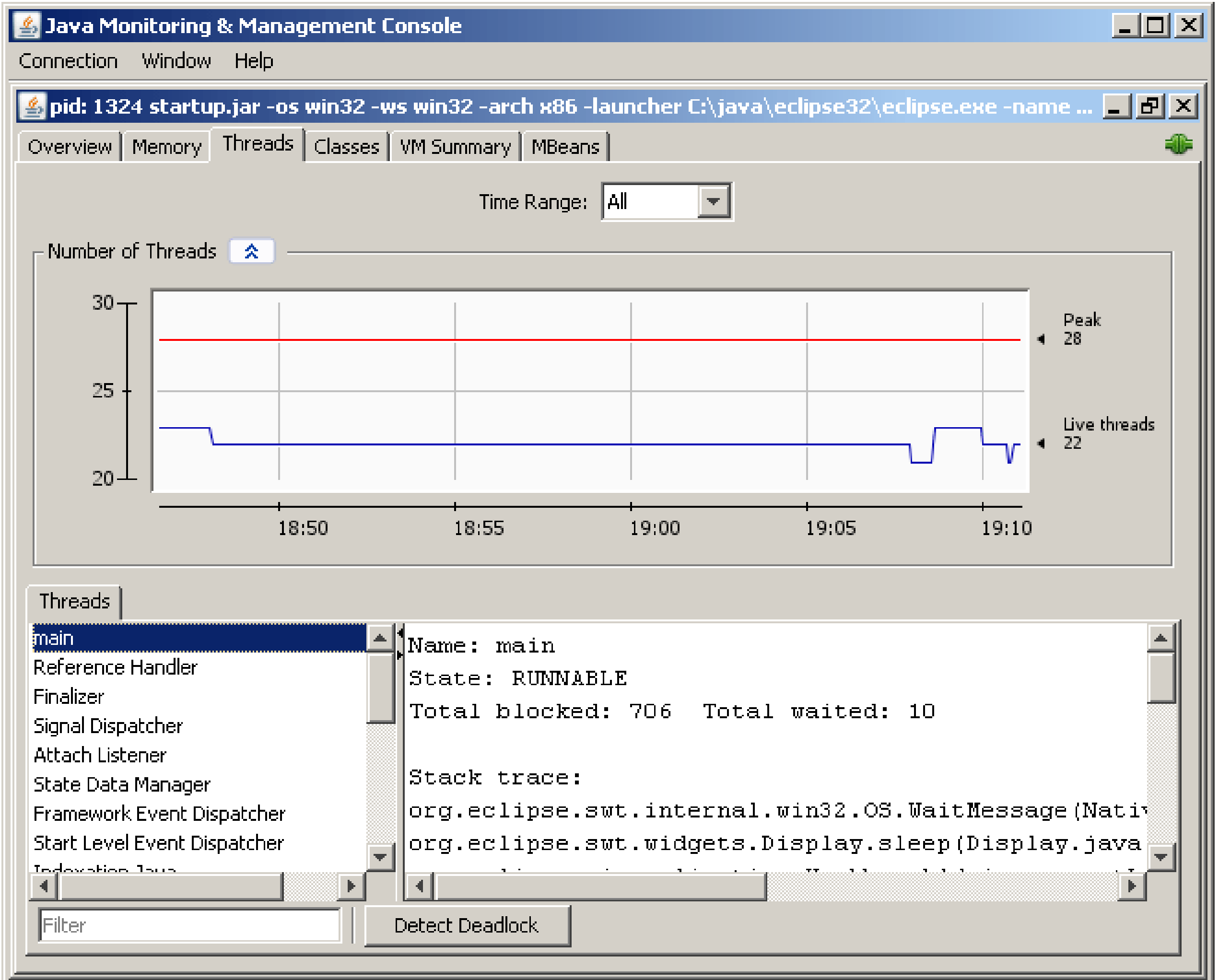
L'onglet « Overview » fourni un résumé graphique sur quatre données : l'usage du tas (heap) et du processeur (CPU) et le nombre de threads et de classes chargées.



L'onglet « Memory » permet d'avoir des informations précises sur la mémoire.

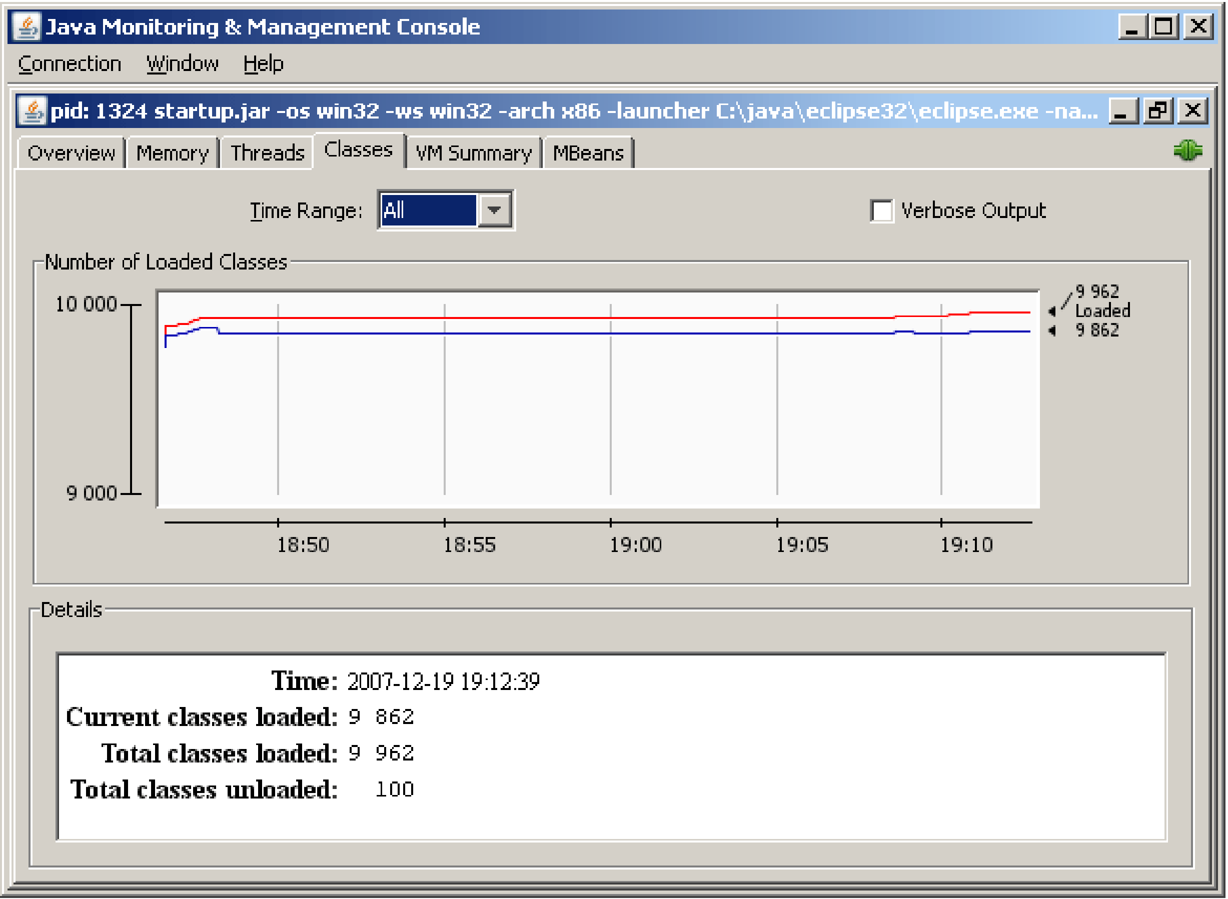
Le contenu du graphique principal et des détails peut être sélectionné dans la liste déroulante chart : Heap Memory Usage, Non-Heap Memory Usage, Eden Spacen Survivor Space, Tenured Space, Code Cache, Perm Gen, Perm Gen shared-rw et Perm Gen shared-ro. Ceci permet d'avoir une vue précise sur chaque partie de la mémoire de la JVM.

Le bouton « Perform GC » permet de demander l'exécution du ramasse miette.

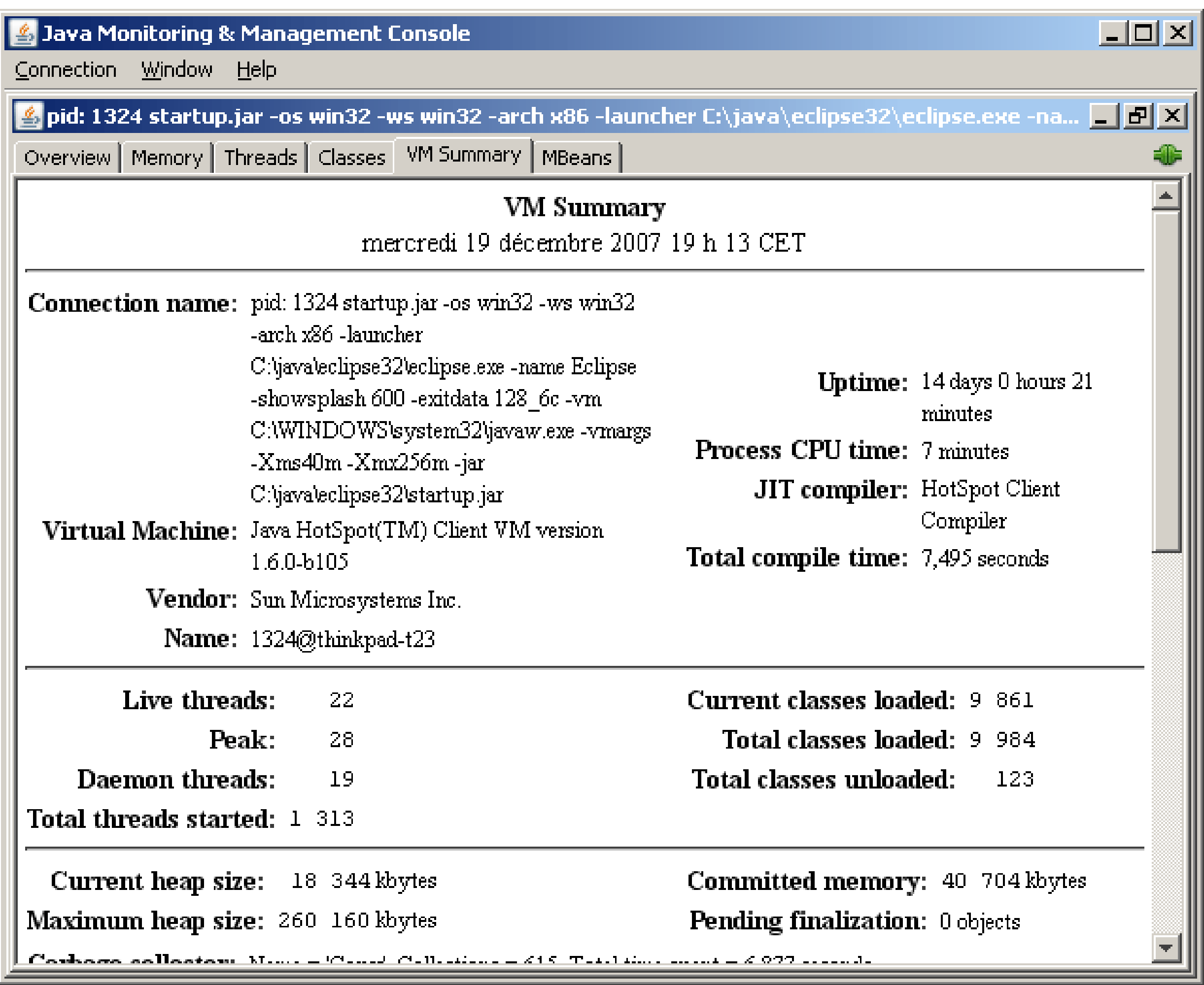


L'onglet « Threads » permet d'avoir des informations sur les threads en cours d'exécution.

Pour avoir des informations précises sur un thread, il suffit de le sélectionner.



L'onglet « Classes » permet de connaître le nombre de classes chargées.



L'onglet « VM Summary » affiche de nombreuses informations sur l'état de la JVM. Celles-ci sont rafraichies périodiquement toutes les 5 secondes.

L'onglet « MBean » permet de gérer les MBeans de l'API JMX qui sont accessibles dans la JVM.

67. JavaDoc

Javadoc est un outil fourni par Sun avec le JDK pour permettre la génération d'une documentation technique à partir du code source.

Cet outil permet la génération d'une documentation au format HTML à partir du code source Java et de commentaires particuliers qu'il contient. Un exemple concret de l'utilisation de cet outil est la documentation du JDK qui est générée grâce à Javadoc.

Il génère une documentation contenant :

- une description détaillée pour chaque classe et ses membres public et protected par défaut (sauf les classes internes anonymes)
- un ensemble de listes (liste des classes, hiérarchie des classes, liste des éléments deprecated et un index général)
- des références croisées et une navigation entre ces différents éléments.

L'intérêt de ce système est de conserver dans le même fichier le code source et les éléments de la documentation qui lui est associé. Il propose donc une auto-documentation des fichiers sources de façon standard.

Ce chapitre contient plusieurs sections :

- La mise en oeuvre
- Les tags définis par javadoc
- Un exemple
- Les fichiers pour enrichir la documentation des packages
- La documentation générée

67.1. La mise en oeuvre

Javadoc s'appuie sur le code source et sur un type de commentaires particuliers pour obtenir des données supplémentaires aux éléments qui composent le code source.

L'outil Javadoc utilise plusieurs types de fichiers source pour générer la documentation :

- Les fichiers sources .java
- Les fichiers de commentaires d'ensemble
- Les fichiers de commentaires des packages
- D'autres fichiers tels que des images, des fichiers HTML, ...

En fonction des paramètres fournis à l'outil, ce dernier recherche les fichiers source .java concernés. Les sources de ces fichiers sont scannées pour déterminer leurs membres, extraire les informations utiles et établir un ensemble de références croisées.

Le résultat de cette recherche peut être enrichi avec des commentaires dédiés insérés dans le code avant chaque élément qu'ils enrichissent. Ces commentaires doivent immédiatement précéder l'entité qu'elle concerne (classe, interface,

méthode, constructeur ou champ). Seul le commentaire qui précède l'entité est traité lors de la génération de la documentation.

Ces commentaires suivent des règles précises. Le format de ces commentaires commence par /** et se termine par */. Il peut contenir un texte libre et des balises particulières.

Le commentaire peut être sur une ou plus généralement sur plusieurs lignes. Les caractères d'espacement (espace et tabulation) qui précède le premier caractère * de chaque ligne du commentaire ainsi que le caractère lui même est ignoré lors de la génération. Ceci permet d'utiliser le caractère * pour aligner le contenu du commentaire.

Exemple :

```
/** Description */
```

Le format général de ces commentaires est :

Exemple :

```
/**
 * Description
 *
 * @tag1
 * @tag2
 */
```

Le commentaire doit commencer par une description de l'élément qui peut utiliser plusieurs lignes. La première phrase de cette description est utilisée par javadoc comme résumé. Cette première phrase se termine par un caractère '.' suivi d'un séparateur (espace ou tabulation ou retour chariot) ou à la rencontre du premier tag Javadoc.

Le texte du commentaire doit être au format HTML : les tags HTML peuvent donc être utilisés pour enrichir le formatage de la documentation. Il est donc aussi nécessaire d'utiliser les entités d'échappement pour certains caractères contenu dans le texte tel que < ou >. Il ne faut surtout pas utiliser les tags de titres <Hn> et le tag du séparateur horizontal <HR> car ils sont utilisés par Javadoc pour structurer le document.

Exemple :

```
/**
 * Description de la classe avec des  <b>mots en gras</b>
 */
```

L'utilisation de balise de formatage HTML est particulièrement intéressante pour formater une description un peu longue en faisant usage notamment du tag <p> pour définir des paragraphes ou du tag <code> pour encadrer un extrait de code.

A partir du JDK 1.4, si la ligne ne commence pas par un caractère *, alors les espaces ne sont plus supprimés (ceci permet par exemple de conserver l'indentation d'un morceau de code contenu dans un tag HTML <PRE>).

Le commentaire peut ensuite contenir des tags Javadoc particuliers qui commencent obligatoirement par le caractère @ et doivent être en début de ligne. Ces tags doivent être regroupés ensemble. Un texte qui suit cet ensemble de tags est ignoré.

Les tags prédéfinis par Javadoc permettent de fournir des informations plus précises sur des composants particuliers de l'élément (auteur, paramètres, valeur de retour, ...). Ces tags sont définis pour un ou plusieurs types d'éléments.

Les tags sont traités de façon particulière par Javadoc. Il existe deux types de tags :

- Block tag : ils sont de la forme @tag
- Inline tag : ils sont de la forme{@tag}

Attention un caractère @ en début de ligne est interprété comme un tag. Si un tel caractère doit apparaître en début de ligne dans la description, il faut utiliser la séquence d'échappement HTML @

Le texte associé à un block tag suit le tag et se termine à la rencontre du tag suivant ou de la fin du commentaire. Ce texte peut donc s'étendre sur plusieurs lignes.
Les tags inline peuvent être utilisés n'importe où dans le commentaire de documentation.

67.2. Les tags définis par javadoc

L'outil Javadoc traite de façon particulière les tags dédiés insérés dans le commentaire de documentation. Javadoc défini plusieurs tags qui permettent de préciser certains composants de l'élément décrit de façon standardisée. Ces tags commencent tous par le caractère arobase @.

Il existe deux types de tags :

- Block tag : ils doivent être regroupés après la description. Ils sont de la forme @tag
- Inline tag : ils peuvent être utilisés n'importe où dans le commentaire. Ils sont de la forme {@tag}

Les block tags doivent obligatoirement débuter en début de ligne (après d'éventuels blancs et un caractère *)

Attention : les tags sont sensibles à la casse.

Pour pouvoir être interprétés, les tags standards doivent obligatoirement commencer en début de ligne.

Tag	Rôle	version du JDK
@author	permet de préciser le ou les auteurs de l'élément	1.0
{@code}		1.5
@deprecated	permet de préciser qu'un élément est déprécié	1.1
{@docRoot}	représente le chemin relatif du répertoire principal de génération de la documentation	1.3
@exception	permet de préciser une exception qui peut être levée par l'élément	1.0
{@inheritDoc}		1.4
{@link}	permet d'insérer un lien vers un élément de la documentation dans n'importe quel texte	1.2
{@linkplain}		1.4
{@literal}		1.5
@param	permet de documenter un paramètre de l'élément	1.0
@return	permet de fournir une description de la valeur de retour d'une méthode qui en possède une	1.0
@see	permet de préciser un élément en relation avec l'élément documenté	1.0
@serial		1.2
@serialData		1.2
@serialField		1.2
@since	permet de préciser depuis quelle version l'élément a été ajouté	1.1
@throws	identique à @exception	1.2
@version	permet de préciser le numéro de version de l'élément	1.0
{@value}		1.4

Ces tags ne peuvent être utilisés que pour commenter certaines entités.

Entité	Tags utilisables
Toutes	@see, @since, @deprecated, {@link}, {@linkplain}, {@docroot}
Overview (fichier overview.html)	@see, @since, @author, @version, {@link}, {@linkplain}, {@docRoot}
Package (fichier package.html)	@see, @since, @serial, @author, @version, {@link}, {@linkplain}, {@docRoot}
Classes et Interfaces	@see, @since, @deprecated, @serial, @author, @version, {@link}, {@linkplain}, {@docRoot}
Constructeurs et méthodes	@see, @since, @deprecated, @param, @return, @throws, @exception, @serialData, {@link}, {@linkplain}, {@inheritDoc}, {@docRoot}
Champs	@see, @since, @deprecated, @serial, @serialField, {@link}, {@linkplain}, {@docRoot}, {@value}

Chacun des tags sera détaillé dans les sections suivantes.

Par convention, il est préférable de regrouper les tags identiques ensemble.

67.2.1. Le tag @author

Le tag @author permet de préciser le ou les auteurs d'une entité.

La syntaxe de ce tag est la suivante :

@author texte

Le texte qui suit la balise est libre. Le doclet standard créé une section "Author" qui contient le texte du tag.

Pour préciser plusieurs auteurs, il est possible d'utiliser un seul ou plusieurs tag @author dans un même commentaire. Dans le premier cas, le contenu du texte est repris intégralement dans la section. Dans le second cas, la section contient le texte de chaque tag séparé par une virgule et un espace.

Exemple :

@author Pierre G.

@author Denis T., Sophie D.

Ce tag n'est utilisable que dans les commentaires d'ensemble, d'une classe ou d'une interface.

A partir du JDK 1.4, il est possible au travers du paramètre -tag de préciser que le tag @author peut être utilisé sur d'autres membres

Exemple :

-tag author:a:"Author:"

67.2.2. Le tag @deprecated

Le tag @deprecated permet de préciser qu'une entité ne devrait plus être utilisée même si elle fonctionne toujours : il permet donc de donner des précisions sur un élément déprécié (deprecated).

La syntaxe de ce tag est la suivante :

@***deprecated*** texte

Il est recommandé de préciser depuis quelle version l'élément est déprécié et de fournir dans le texte libre une description de la solution de remplacement si elle existe ainsi qu'un lien vers une entité de substitution.

Le doclet standard créé une section "Deprecated" avec l'explication dans la documentation.

Remarque : Ce tag est particulier car il est le seul reconnu par le compilateur : celui ci prend note de cet attribut lors de la compilation pour permettre d'en informer les utilisateurs. Lors de la compilation, l'utilisation d'entité marquée avec le tag @deprecated générera un avertissement (warning) de la part du compilateur.

Exemple Java 1.1 :

@deprecated Remplacé par setMessage

@see #setMessage

Exemple Java 1.2 :

@deprecated Remplaçé par {@link #setMessage}

67.2.3. Le tag @exception et @throws

Ce tag permet de documenter une exception levée par la méthode ou le constructeur décrit par le commentaire.

Syntaxe :

@***exception*** nom_exception description

Les tags @exception et throws sont similaires.

Ils sont suivis du nom de l'exception suivi d'une courte description des raisons de la levée de cette dernière. Il faut utiliser autant de tag @exception ou @throws que d'exceptions. Ce tag doit être utilisé uniquement pour un élément de type méthode.

Il ne faut pas mettre de séparateur particulier comme un caractère '-' entre le nom et la description puisque l'outil en ajoute un automatiquement. Il est cependant possible d'aligner les descriptions de plusieurs paramètres en utilisant des espaces afin de faciliter la lecture.

Exemple :

@exception java.io.FileNotFoundException le fichier n'existe pas

Le doclet standard créé une section "Throws" qui regroupe les tags @param du commentaire. L'outil recherche le nom pleinement qualifié de l'exception si c'est simplement son nom est précisé dans le tag.

Exemple extrait de la documentation de l'API du JDK :

`public **String**(char[] value)`

Allocates a new `String` so that it represents the sequence of characters currently contained in the character array argument. The contents of the character array are copied; subsequent modification of the character array does not affect the newly created string.

Parameters:

`value` - the initial value of the string.

Throws:

`NullPointerException` - if `value` is `null`.

67.2.4. Le tag @param

Le tag @param permet de documenter un paramètre d'une méthode ou d'un constructeur. Ce tag doit être utilisé uniquement pour un élément de type constructeur ou méthode.

La syntaxe de ce tag est la suivante :

@param nom_paramètre description du paramètre

Ce tag est suivi du nom du paramètre (ne pas utiliser le type) puis d'une courte description de ce dernier. A partir de Java 5, il est possible d'utiliser le type du paramètre entre les caractères < et > pour une classe ou une méthode.

Il ne faut pas mettre de séparateur particulier comme un caractère '-' entre le nom et la description puisque l'outil en ajoute un automatiquement. Il est cependant possible d'aligner les descriptions de plusieurs paramètres en utilisant des espaces afin faciliter la lecture.

Il faut utiliser autant de tag @param que de paramètres dans la signature de l'entité concernée. La description peut être contenue sur plusieurs lignes.

Le doclet standard créé une section "Parameters" qui regroupe les tags @param du commentaire. Il génère pour chaque tag une ligne dans cette section avec son nom et sa description dans la documentation.

Exemple extrait de la documentation de l'API du JDK :

`public` **`String`**`(String value)`

Initializes a newly created `String` object so that it represents the same sequence of characters as the argument; in other words, the newly created string is a copy of the argument string.

Parameters:

`value` - a `String`.

Par convention les paramètres doivent être décrits dans leur ordre dans la signature de la méthode décrite

Exemple :

@param nom nom de la personne

@param message chaine de caractères à traiter. Si cette valeur est <code>null</code> alors une exception est levée

Exemple 2 : /** * @param <E> Type des elements stockés dans la collection */
public interface List<E> extends Collection<E> { }

67.2.5. Le tag @return

Le tag @return permet de fournir une description de la valeur de retour d'une méthode qui en possède une.

La syntaxe de ce tag est la suivante :

@return description_de_la_valeur_de retour_de_la_méthode

Il ne peut y avoir qu'un seul tag @return par commentaire : il doit être utilisé uniquement pour un élément de type méthode qui renvoie une valeur.

Avec le doclet standard, ce tag créé une section "Returns" qui contient le texte du tag. La description peut tenir sur plusieurs lignes.

Exemple extrait de la documentation de l'API du JDK :

getClass

```
public final Class getClass()
```

Returns the runtime class of an object. That `Class` object is the object that is locked by `static synchronized` methods of the represented class.

Returns:

the object of type `Class` that represents the runtime class of the object.

Il ne faut pas utiliser ce tag pour des méthodes ne possédant pas de valeur de retour (void).

Exemple:

@return le nombre d'occurrences contenues dans la collection

@return <code>true</code> si les traitements sont correctement exécutés sinon <code>false</code>

67.2.6. Le tag @see

Le tag @see permet de définir un renvoi vers une autre entité incluse dans une documentation de type Javadoc ou vers une url.

La syntaxe de ce tag est la suivante :

@see référence à une entité suivie d'un libellé optionnel ou lien ou texte entre double quote

@see package
@see package.Class
@see class
@see #champ
@see class#champ
@see #method(Type,Type,...)
@see class#method(Type,Type,...)
@see package.class#method(Type,Type,...)
@see ...
@see " ... "

Le tag génère un lien vers une entité ayant un lien avec celle documentée.

Il peut y avoir plusieurs tags @see dans un même commentaire.

L'entité vers laquelle se fait le renvoi peut être un package, une classe, une méthode ou un lien vers une page de la documentation. Le nom de la classe doit être de préférence pleinement qualifié.

Le caractère # permet de séparer une classe d'un de ces membres (champ, constructeur ou méthode). Attention : il ne faut surtout pas utiliser le caractère "." comme séparateur entre une classe ou une interface et le membre précisé.

Pour préciser une version surchargée précise d'une méthode ou d'un constructeur, il suffit de préciser la liste des types d'arguments de la version concernée.

Il est possible de fournir un libellé optionnel à la suite de l'entité. Ce libellé sera utilisé comme libellé du lien généré : ceci est pratique pour forcer un libellé à la place de celui généré automatiquement (par défaut le nom de l'entité).

Si le tag est suivi d'un texte entre double cote, le texte est simplement repris avec les cotes sans lien.

Si le tag est suivi d'un tag HTML <a>, le lien proposé par ce tag est repris intégralement.

Le doclet standard créé une section "See Also" qui regroupe les tags @see du commentaire en les séparant par une virgule et un espace.

Exemple extrait de la documentation de l'API du JDK :

```
public static String valueOf(Object obj)
```

Returns the string representation of the `Object` argument.

Parameters:

`obj` - an `Object`.

Returns:

if the argument is `null`, then a string equal to `"null"`; otherwise, the value of `obj.toString()` is returned.

See Also:

`Object.toString()`

Remarque : pour insérer un lien n'importe où dans le commente, il faut utiliser le tag {@link}

Exemple :

@see String
@see java.lang.String
@see String#equals
@see java.lang.Object#wait(int)
@see MaClasse nouvelle classe
@see Test
@see "Le dossier de spécification détaillée"

Ce tag permet de définir des liens vers d'autres éléments de l'API.

67.2.7. Le tag @since

Le tag @since permet de préciser un numéro de version de la classe ou de l'interface à partir de laquelle l'élément décrit est disponible. Ce tag peut être utilisé avec tous les éléments.

La syntaxe de ce tag est la suivante :

@since texte

Le texte qui représente le numéro de version est libre. Le doclet standard créé une section "Since" qui contient le texte du tag.

Exemple extrait de la documentation de l'API du JDK :

```
public byte[] getBytes()
```

Convert this `String` into bytes according to the platform's default character encoding, storing the result into a new byte array.

Returns:

the resultant byte array.

Since:

JDK1.1

Par convention pour limiter le nombre de section Since dans la documentation, lorsqu'une nouvelle classe ou interface est ajoutée, il est préférable de mettre un tag @since sur le commentaire de la classe et ne pas le reporter sur chacun ses membres. Le tag @since est utilisé sur un membre uniquement lors de l'ajout du membre.

Dans la documentation de l'API Java, ce tag préciser depuis qu'elle version du JDK l'entité décrite est utilisable.

Exemple :

@since 2.0

67.2.8. Le tag @version

Le tag @version permet de préciser un numéro de version. Ce tag doit être utilisé uniquement pour un élément de type classe ou interface.

La syntaxe de ce tag est la suivante :

@version texte

Le texte qui suit la balise est libre : il devrait correspondre à la version courante de l'entité documentée. Le doclet standard créé une section "Version" qui contient le texte du tag.

Il ne devrait y avoir qu'un seul tag @version dans un commentaire.

Par défaut, le doclet standard ne prend pas en compte ce tag : il est nécessaire de demander sa prise en compte avec l'option -version de la commande javadoc.

Exemple :

@version 1.00

67.2.9. Le tag {@link}

Ce tag permet de créer un lien vers un autre élément de la documentation.

La syntaxe de ce tag est la suivante :

{@***link*** package.class#membre texte }

Le mode de fonctionnement de ce tag est similaire au tag @see : la différence est que le tag @see créé avec le doclet standard un lien dans la section "See also" alors que le tag {@link} créé un lien à n'importe quel endroit de la documentation.

Si une accolade fermante doit être utilisée dans le texte du tag il faut utiliser la séquence d'échappement }.

Exemple :

Utiliser la {@link #maMethode(int) nouvelle méthode}

67.2.10. Le tag {@value}

Ce tag permet d'afficher la valeur d'un champ.

La syntaxe de ce tag est la suivante :

{@value}

{@value package.classe#champ_static}

Lorsque le tag {@value} est utilisé sans argument avec un champ static, le tag est remplacé par la valeur du champ.

Lorsque le tag {@value} est utilisé avec comme argument une référence à un champ static, le tag est remplacé par la valeur du champ précisé. La référence utilisée avec ce tag suit la même forme que celle du tag @see

Exemple :

{@value}

{@value #MA_CONSTANTE}

67.2.11. Le tag {@literal}

Ce tag permet d'afficher un texte qui ne sera pas interprété comme de l'HTML.

La syntaxe de ce tag est la suivante :

{@***literal*** texte}

Le contenu du texte est repris intégralement sans interprétation. Notamment les caractères < et > ne sont pas interprétés comme des tags HTML.

Pour afficher du code, il est préférable d'utiliser le tag {@code}

Exemple :

{@literal 0<b>10}

67.2.12. Le tag {@linkplain}

Ce tag permet de créer un lien vers un autre élément de la documentation dans une police normale.

Ce tag est similaire au tag @link. La différence réside dans la police d'affichage.

67.2.13. Le tag {@inheritDoc}

Ce tag permet de demander explicitement la recopie de la documentation de l'entité de la classe mère la plus proche correspondante.

La syntaxe de ce tag est la suivante:

{@inheritDoc}

Ce tag permet d'éviter le copier/coller de la documentation d'une entité.

Il peut être utilisé :

- dans la description d'une entité : dans ce cas tout le commentaire de l'entité de la classe mère est repris
- dans un tag @return, @tag, @throws : dans ce cas tout le texte du tag de l'entité de la classe mère est repris

67.2.14. Le tag {@docRoot}

Ce tag représente le chemin relatif par rapport à la documentation générée.

La syntaxe de ce tag est la suivante :

{@docRoot}

Ce tag est pratique pour permettre l'inclusion de fichiers dans la documentation.

Exemple :

Historique

67.2.15. Le tag {@code}

Ce tag permet d'afficher un texte dans des tags <code> ... </code> qui ne sera pas interprété comme de l'HTML.

La syntaxe de ce tag est la suivante :

{@code texte}

Le contenu du texte est repris intégralement sans interprétation. Notamment les caractères < et > ne sont pas interprétés comme des tags HTML.

Le tag {@code texte} est équivalent à <code>{@literal texte}</code>

Exemple :

{@code 0<b>10}

67.3. Un exemple

Exemple :

```
/**
 * Résumé du rôle de la methode.
 * Commentaires détaillées sur le role de la methode
 * @param val la valeur a traiter
 * @return la valeur calculée
 * @since 1.0
 * @deprecated Utiliser la nouvelle methode XXX
 */
  public int maMethode(int val) {
    return 0;
  }
```

Résultat :

---

maMethode

```
public int maMethode(int val)
```

Deprecated. *Utiliser la nouvelle methode xyz*

Résumé du rôle de la methode. Commentaires détaillées sur le role de la methode

Parameters:
`val` - la valeur a traiter
Returns:
la valeur calculée
Since:
1.0

67.4. Les fichiers pour enrichir la documentation des packages

Javadoc permet de fournir un moyen de documenter les packages car ceux ci ne disposent pas de code source particulier : il faut définir des fichiers dont le nom est particulier.

Ces fichiers doivent être placés dans le répertoire désigné par le package.

Le fichier package.html contient une description du package au format HTML. En plus, il est possible d'utiliser les tags @deprecated, @link, @see et @since.

Le fichier overview.html permet de fournir un résumé de plusieurs packages au format html. Ce fichier doit être placé dans le répertoire qui inclus les packages décrits.

67.5. La documentation générée

Pour générer la documentation, il faut invoquer l'outil javadoc. Javadoc recrée à chaque utilisation la totalité de la documentation.

Pour formater la documentation, javadoc utilise une doclet. Une doclet permet de préciser le format de la documentation générée. Par défaut, Javadoc propose une doclet qui génère une documentation au format HTML. Il est possible de définir sa propre doclet pour changer le contenu ou le format de la documentation (pour par exemple, générer du RTF ou du XML).

La génération de la documentation avec le doclet par défaut créé de nombreux fichiers et des répertoires pour structurer la documentation au format HTML avec et sans frame.

La documentation de l'API Java fourni par Sun est réalisée grâce à Javadoc. La page principale est composée de trois frames :

Par défaut, la documentation générée contient les éléments suivants :

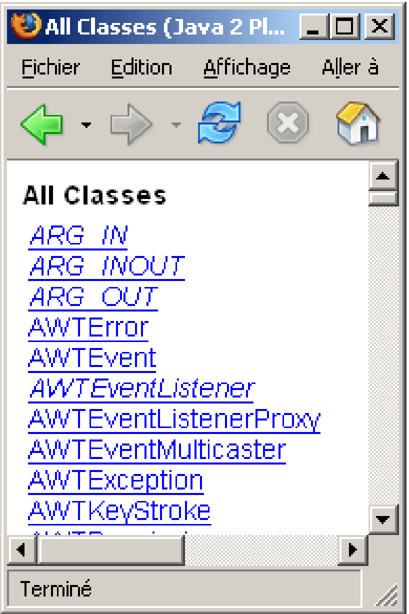
- un fichier html par classe ou interface qui contient le détail chaque élément de la classe ou interface
- un fichier html par package qui contient un résumé du contenu du package
- un fichier overview-summary.html
- un fichier overview-tree.html
- un fichier deprecated-list.html
- un fichier serialized-form.html
- un fichier overview-frame.html
- un fichier all-classe.html
- un fichier package-summary.html pour chaque package
- un fichier package-frame.html pour chaque package
- un fichier package-tree.html pour chaque package

Tous ces fichiers peuvent être regroupés en trois catégories :

- Les pages de base : les pages des classes et interfaces et les résumés
- Les pages des références croisés : les pages index, les pages de hiérarchie, les pages d'utilisation, et les pages deprecated-list.html, constant-values.html et serialized-form.html
- Les fichiers de structure : la page principale, les frames, la feuille de style

Il y a plusieurs fichiers générés à la racine de l'application :

Le fichier allclasses-frame.html affiche toutes les classes, interfaces et exception de la documentation avec un lien pour afficher le détail. Cette page est affichée en bas à gauche dans le fichier index.html



Le fichier constant-values.html affiche la liste de toutes les constantes avec leur valeur

Constant Field Values (Java 2 Platform SE v1.4.2) - Mozilla Firefox

Fichier Edition Affichage Aller à Marque-pages Outils ?

file:///C:/j2sdk1.4.2_04/docs/api/constant-values.html

Overview Package Class Use Tree Deprecated Index Help

Java™ 2 Platform Std. Ed. v1.4.2

PREV NEXT FRAMES NO FRAMES All Classes

Constant Field Values

Contents

- java.awt.*
- java.beans.*
- java.io.*

Terminé

Constant Field Values (Java 2 Platform SE v1.4.2) - Mozilla Firefox

Fichier Edition Affichage Aller à Marque-pages Outils ?

file:///C:/j2sdk1.4.2_04/docs/api/constant-values.html

java.awt.*

java.awt.Adjustable

```
public static final int HORIZONTAL     0
public static final int NO_ORIENTATION 2
public static final int VERTICAL       1
```

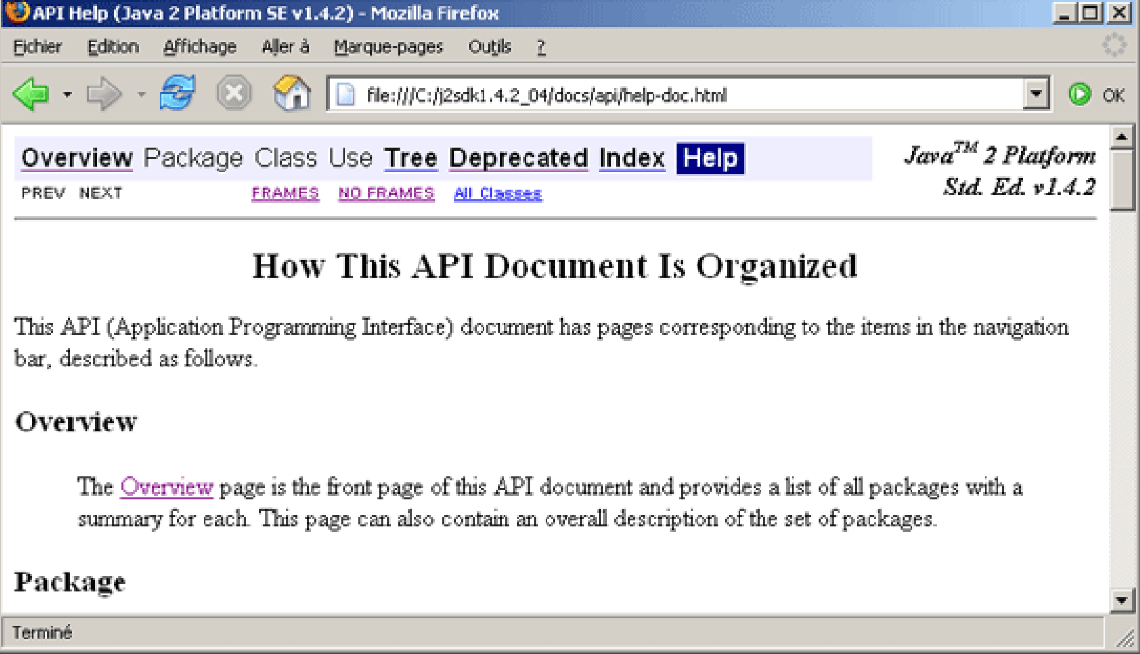
java.awt.AlphaComposite

```
public static final int CLEAR     1
```

Terminé

Le fichier deprecated-list.html affiche la liste de tous les membres déclarés deprecated. Le lien Deprecated de la barre de navigation permet d'afficher le contenu de cette page.

Le fichier help-doc.html affiche l'aide en ligne de la documentation. Le lien Help de la barre de navigation permet d'afficher le contenu de cette page.

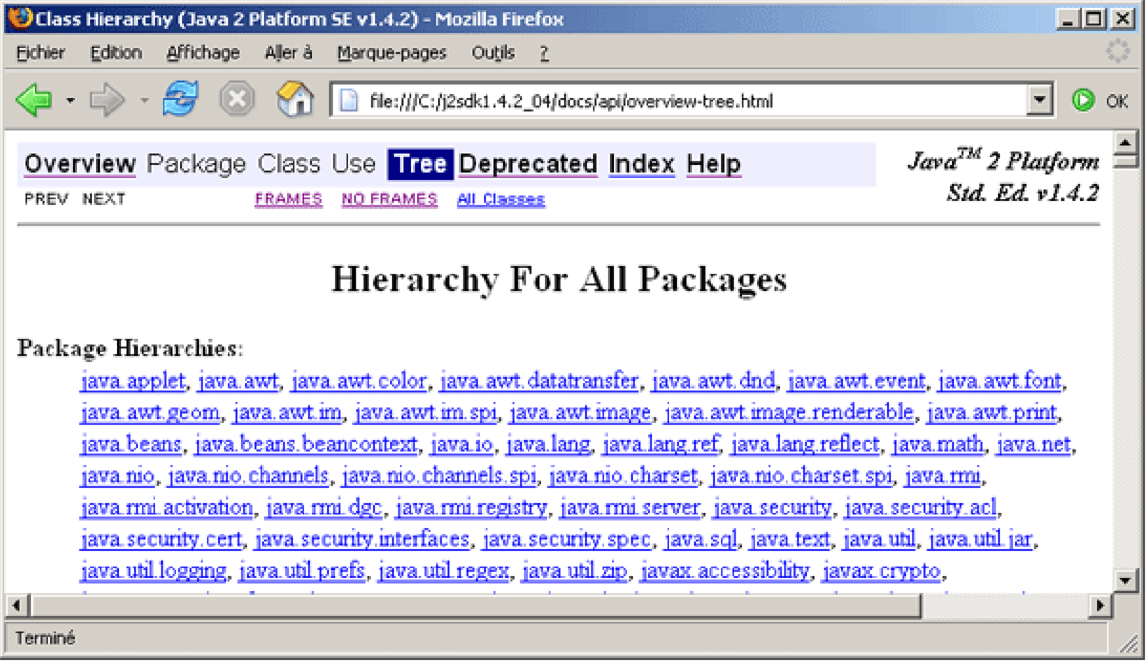


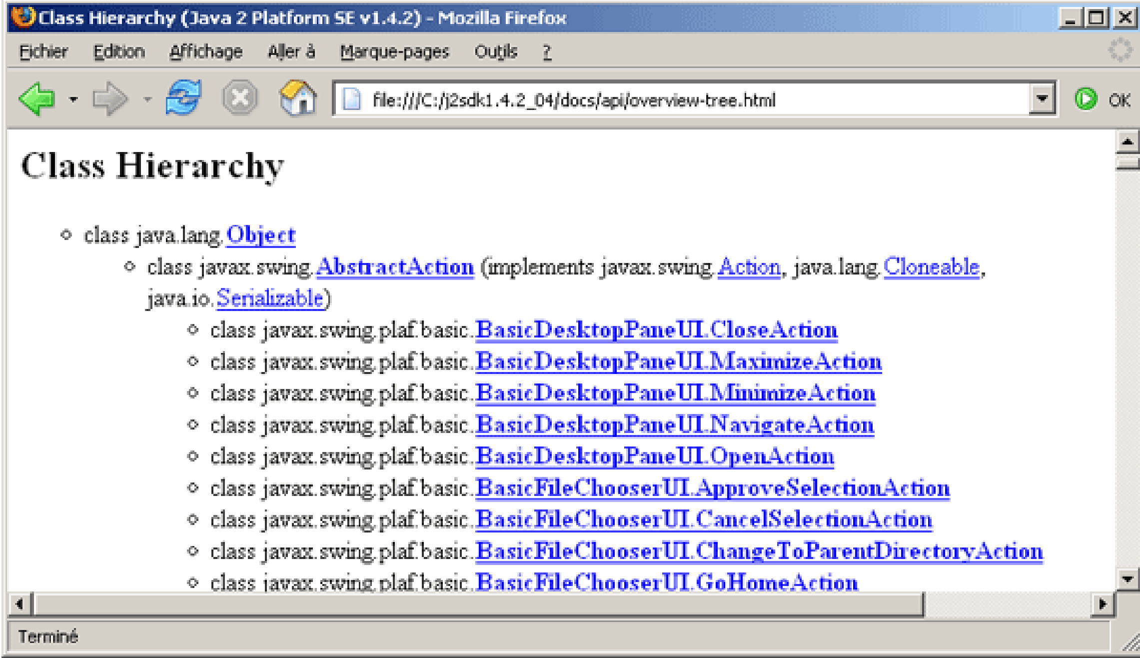
Le fichier index.html est la page principale de la documentation composée de 3 frames

Le fichier overview-frame.html affiche la liste des packages avec un lien pour afficher la liste membres du package. Cette page est affichée en haut à gauche dans le fichier index.html

Le fichier overview-summary.html affiche un résumé des packages de la documentation. Cette page est affichée par défaut dans la partie centrale de la page index.html

Le fichier overview-tree.html affiche la hiérarchie des classes et interfaces. Le lien Tree de la barre de navigation permet d'afficher le contenu de cette page.

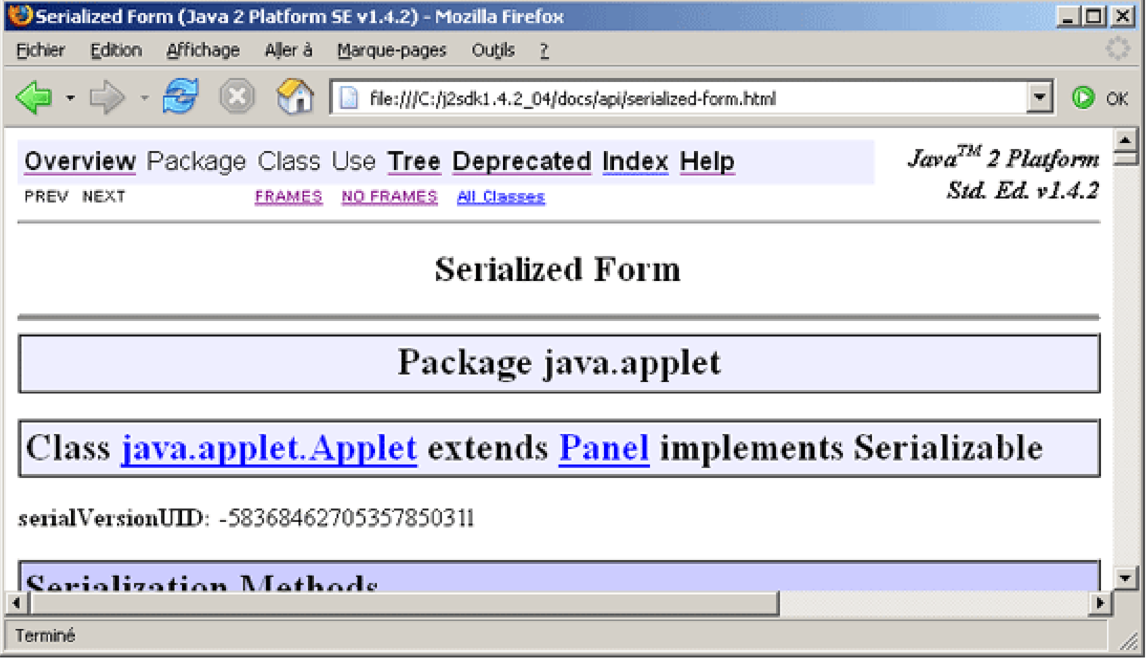




Le fichier package-list est un fichier texte contenant la liste de tout les packages (non affiché dans la documentation).

Le fichier packages.html permet une choisir entre la version avec et sans frame de la documentation

Le fichier serialized-form.html affiche la liste des classes qui sont sérialisables



Le fichier stylesheet.css est la feuille de style utilisée pour afficher la documentation.

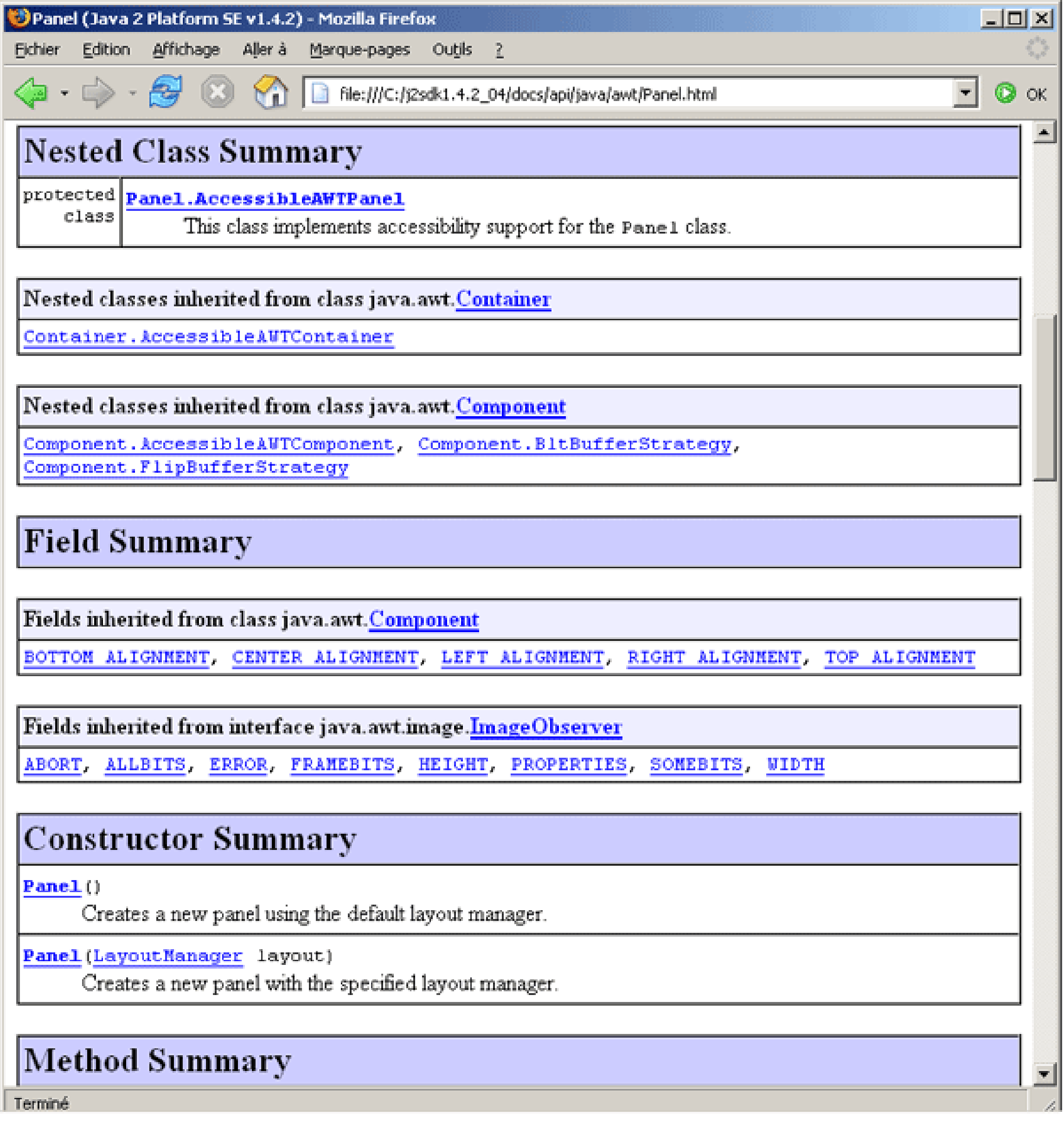
Le fichier allclasses-noframe.html affiche la page allclasses-frame.html sans frame.

Il y a un répertoire par package. Ce répertoire contient plusieurs fichiers :

- classe.html : un fichier html pour chaque classe du package contenant sa documentation
- package-frame.html : contient la liste de toutes les interfaces, classes et exceptions du package
- package-summary.html : contient un résumé de toutes les interfaces, classes et exceptions du package
- package-tree.html : contient l'arborescence de toutes les interfaces et classes du package

Cette structure est reprise pour les sous packages.

La page détaillant une classe possède la structure suivante :


Panel (Java 2 Platform SE v1.4.2) - Mozilla Firefox

Fichier Edition Affichage Aller à Marque-pages Outils ?

file:///C:/j2sdk1.4.2_04/docs/api/java/awt/Panel.html

Nested Class Summary

protected class	Panel.AccessibleAWTPanel This class implements accessibility support for the Panel class.

Nested classes inherited from class java.awt.Container
Container.AccessibleAWTContainer

Nested classes inherited from class java.awt.Component
Component.AccessibleAWTComponent, Component.BltBufferStrategy, Component.FlipBufferStrategy

Field Summary

Fields inherited from class java.awt.Component
BOTTOM_ALIGNMENT, CENTER_ALIGNMENT, LEFT_ALIGNMENT, RIGHT_ALIGNMENT, TOP_ALIGNMENT

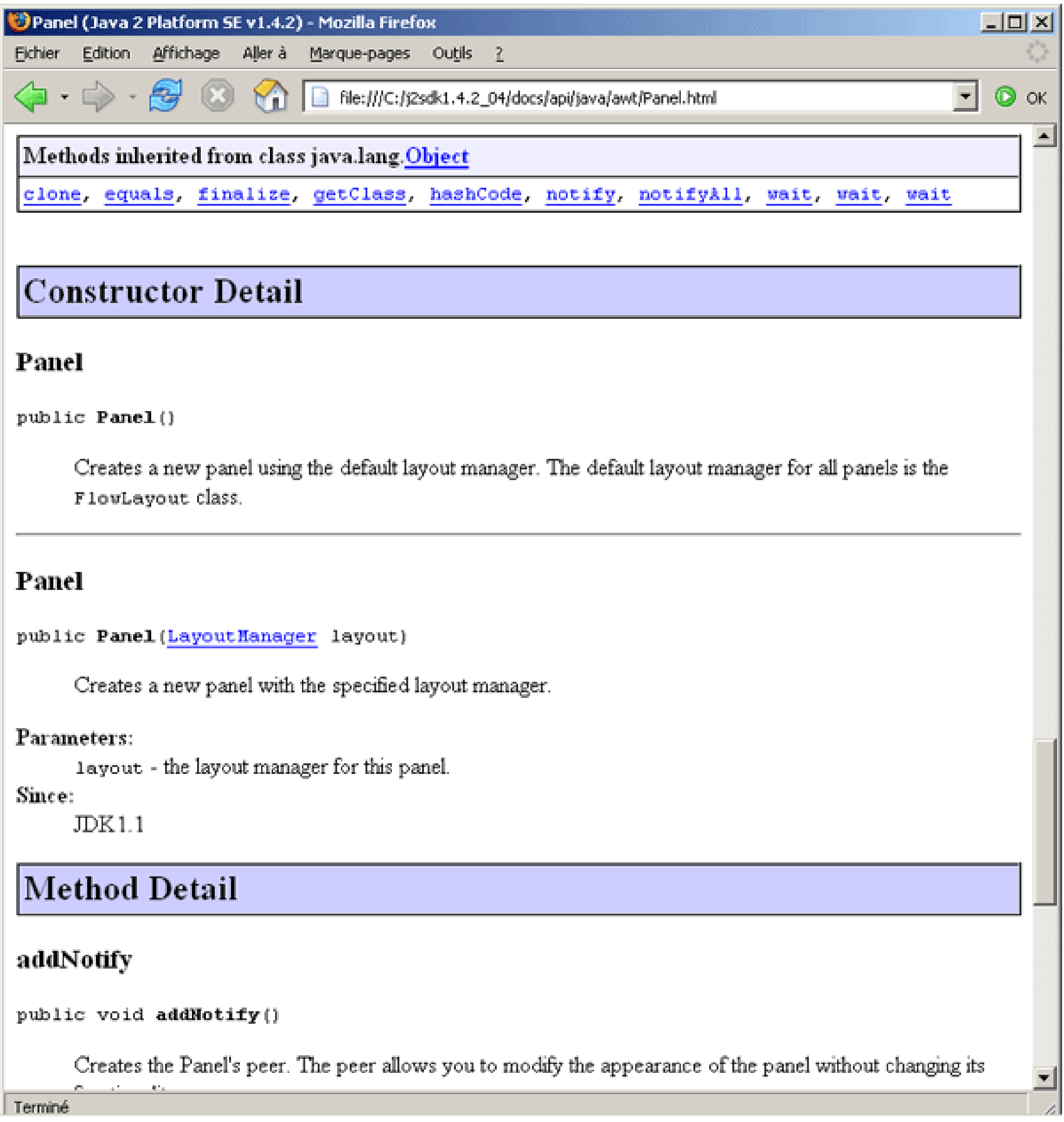
Fields inherited from interface java.awt.image.ImageObserver
ABORT, ALLBITS, ERROR, FRAMEBITS, HEIGHT, PROPERTIES, SOMEBITS, WIDTH

Constructor Summary

Panel() Creates a new panel using the default layout manager.
Panel(LayoutManager layout) Creates a new panel with the specified layout manager.

Method Summary

Terminé



Si l'option -linksource est utilisée, les fichiers sources sont stockés dans l'arborescence du sous répertoire src-html de la documentation.

68. Les outils libres et commerciaux

Pour développer des composants en java (applications clientes, applets, applications web, services web, ...), il existe une large gamme d'outils commerciaux et libres pour répondre à ce vaste marché.

Comme dans d'autres domaines, les avantages et les inconvénients de ces outils sont semblables selon leur catégorie bien qu'ils ne puissent pas être complètement généralisés :

	Avantages	Inconvénient
Outils commerciaux	une meilleure ergonomie une hot line dédiée	le prix
Outils libres	la gratuité des mises à jour fréquentes (variable selon le projet)	pas de support officiel (aide communautaire via les forums)

Certains de ces outils libres n'ont que peu de choses à envier à certains de leurs homologues commerciaux : ainsi Tomcat du projet Jakarta est l'implémentation de référence pour ce qui concerne les servlets et les JSP.

Enfin certains éditeurs, surtout dans le domaine des IDE, proposent souvent une version limitée (dans les fonctionnalités ou dans le temps) mais gratuite qui permet d'utiliser et d'évaluer le produit.

L'évolution des ces outils suit l'évolution du marché concernant java : développement d'applet (web client), d'application autonome et C/S, et maintenant développement côté serveur (applications et services web).

La liste des produits de ce chapitre est loin d'être exhaustive mais représente les plus connus ou ceux que j'utilise.

Ce chapitre contient plusieurs sections :

- Les environnements de développements intégrés (IDE)
- Les serveurs d'application
- Les conteneurs web
- Les conteneurs d'EJB
- Les outils divers
- Les MOM
- Les outils pour bases de données
- Les outils de modélisation UML

68.1. Les environnements de développements intégrés (IDE)

Les environnements de développements intégrés regroupent dans un même outil la possibilité d'écrire du code source, de concevoir une application de façon visuelle par assemblage de beans, d'exécuter et de débugger le code.

D'une façon générale, ils sont tous très gourmands en ressources machines : un processeur rapide, 256 Mo de RAM pour être à l'aise ... En fait la plupart de ces outils sont partiellement ou totalement écrits en Java.

Le choix d'un IDE doit tenir compte de plusieurs caractéristiques : ergonomie et convivialité pour faciliter l'utilisation, fonctionnalités de bases et avancées pour accroître la productivité, robustesse, support des standards, ... Tous les éditeurs proposent une version libre qui permet d'évaluer leur produit.

68.1.1. Le projet Eclipse

Eclipse est un projet open source à l'origine développé par IBM pour ses futurs outils de développement et offert à la communauté. Le but est de fournir un outil modulaire capable non seulement de faire du développement en Java mais aussi dans d'autres langages et d'autres activités. Cette polyvalence est liée au développement de modules (plug-in) réalisés par la communauté ou des entités commerciales.

La fondation Eclipse gère de nombreux sous projets :

Projet	Description
Eclipse	ce projet développe l'architecture et la structure de la plate-forme Eclipse.
Eclipse Tools	ce projet développe ou intègre des outils à la plate-forme pour permettre à des tiers d'enrichir la plate-forme. Il possède plusieurs sous projets tel que CDT (plug-in pour le développement en C/C++), AspectJ (AOP), GEF (Graphical Editing Framework), PHP (plug-in pour le développement en PHP), Cobol (plug-in pour le développement en Cobol), VE (Visual Editor) pour la création d'IHM.
Eclipse Technology	ce projet, divisé en trois catégories, propose d'effectuer des recherches sur des évolutions de la plate-forme et des technologies qu'elle met en oeuvre.
Web Tools Platform (WTP)	ce projet a pour but d'enrichir la plate-forme enfin de proposer un framework et des services pour la création d'outils de développement d'applications web. Il est composé de plusieurs sous projets : WST (Web Standard Tools), JST (J2EE Standard Tools), ATF (Ajax Toolkit Framework), Dali (mapping avec JPA) et JSF (Java Server Faces)
Test and Performance Tools Platform (TPTP)	ce projet a pour but de développer une plate-forme servant de support à la création d'outils de tests et d'analyses
Business Intelligence and Reporting Tools (BIRT)	ce projet a pour but de développer une plate-forme facilitant l'intégration de générateur d'états. Il est composé de 4 sous projets : ERD (Eclipse Report Designer), WRD (Web based Report Designer), ERE (Eclipse Report Engine) et ECE (Eclipse Charting Engine).
Eclipse Modeling	ce projet contient plusieurs sous projet dont EMF (Eclipse Modeling Framework) et UML2 pour une implémentation d'UML reposant sur EMF
Data Tools Platform (DTP)	ce projet a pour but de manipuler des sources de données (bases de données relationnelles)
Device Software Development Platform	ce projet a pour but de créer des plug-ins pour faciliter le développement d'applications sur appareils mobiles
Eclipse SOA Tools Platform	ce projet a pour but de développer des outils pour faciliter la mise en d'architecture de type SOA

Le site officiel est à l'url www.eclipse.org/

Bien que développé en Java, les performances à l'exécution d'Eclipse sont très bonnes car il n'utilise pas Swing pour l'interface homme-machine mais un toolkit particulier nommé SWT associé à la bibliothèque JFace. SWT (Standard Widget Toolkit) est développé en Java par IBM en utilisant au maximum les composants natifs fournis par le système d'exploitation sous jacent. JFace utilise SWT et propose une API pour faciliter le développement d'interfaces graphiques.

Eclipse ne peut donc fonctionner que sur les plateformes pour lesquelles SWT a été porté.

SWT et JFace sont utilisés par Eclipse pour développer le plan de travail (Workbench) qui organise la structure de la plate-forme et les interactions entre les outils et l'utilisateur. Cette structure repose sur trois concepts : la perspective, la vue et l'éditeur. La perspective regroupe des vues et des éditeurs pour offrir une vision particulière des développements. En standard, Eclipse propose huit perspectives.

Les vues permettent de visualiser et de sélectionner des éléments. Les éditeurs permettent de visualiser et de modifier le contenu d'un élément de l'espace de travail.

La version 3.3 de cet outil est diffusée en juillet 2007.

68.1.2. IBM Websphere Studio Application Developper

Websphere Studio Application Developper (WSAD) représente le nouvel outil de développement d'applications Java/web d'IBM. Il représente une fusion de nombreuses fonctionnalités des outils Visual Age for Java et Websphere Studio. Le coeur de l'outil est composé par Websphere Studio Workbench dont une partie du code a été fournie à la communauté open source pour devenir le projet Eclipse.

www-4.ibm.com/software/ad/studioappdev/

WSAD version 4.0 est orienté développement Java/web : il ne permet pas de développement d'applications graphiques en mode RAD.

WSAD est remplacé par le produit Rational Application Developer for WebSphere Software

68.1.3. IBM Rational Application Developer for WebSphere Software

Rational Application Developer for WebSphere Software repose sur la version 3.2 d'Eclipse et propose le développement d'applications web et portail ou d'entreprise ou standalone, de services web pour mettre en oeuvre une SOA. Cet outil s'intègre parfaitement avec les outils IBM et Rational.

http://www-306.ibm.com/software/awdtools/developer/application/

68.1.4. MyEclipse

MyEclipse est un IDE basé sur Eclipse développé par Genuitec. Le site officiel du produit est à l'url www.myeclipseide.com/

MyEclipse regroupe de nombreux plug-ins dont certains sont inédits comme par exemple le portage de Matisse de NetBeans sur Eclipse.

La version courante est la 5.1.

68.1.5. Netbeans

Netbeans est un environnement de développement en java open source écrit en java. Le produit est

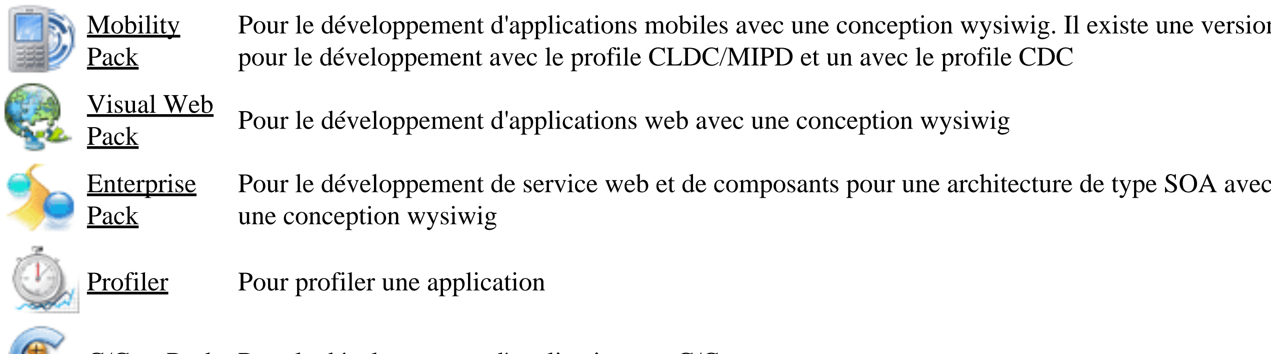
composé d'une partie centrale à laquelle il est possible d'ajouter des modules.

Netbeans est un IDE open source développé par Sun Microsystems. Il est téléchargeable gratuitement sur le site officiel du produit www.netbeans.org.

Netbeans propose des fonctionnalités permettant le développement d'applications standalone (AWT/Swing), web (Servlets, JSP, Struts, JSF), mobile (J2ME) ou d'entreprise (J2EE/JEE).

La version courante est la 5.5 : il est peut être téléchargée seul ou packagé avec un JDK (5.0 ou 6.0) ou avec un serveur d'application (Java EE Application Server 9.0 JBoss AS 4.0.4). Il fonctionne sous Windows, Linux, Mac OS X, et Solaris.

Netbeans est modulaire et propose plusieurs plug-ins officiels :

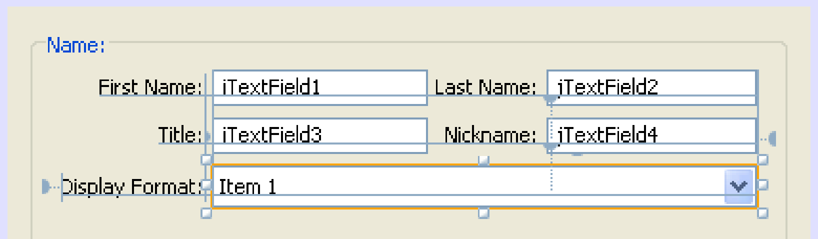


Plug-in	Description
Mobility Pack	Pour le développement d'applications mobiles avec une conception wysiwig. Il existe une version pour le développement avec le profile CLDC/MIPD et un avec le profile CDC
Visual Web Pack	Pour le développement d'applications web avec une conception wysiwig
Enterprise Pack	Pour le développement de service web et de composants pour une architecture de type SOA avec une conception wysiwig
Profiler	Pour profiler une application
C/C++ Pack	Pour le développement d'applications en C/C++

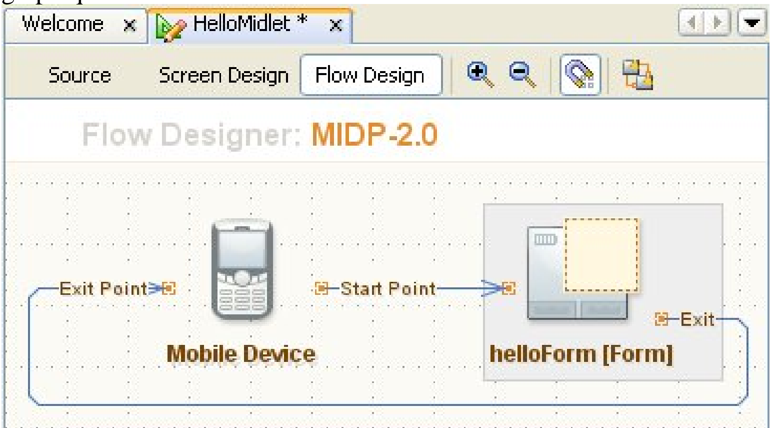
Il existe aussi de nombreux plug-ins développés par des tiers (une liste peut être consultée à l'url www.netbeans.org/catalogue/index.html)

Quelques fonctionnalités de Netbeans sont particulièrement intéressantes :

- Le développement wysiwig d'applications reposant sur Swing (projet Matisse) qui propose un positionnement aisé des composants



- Le plug-in Mobility pack qui facilite le développement des midlets en gérant leurs enchainements graphiquement



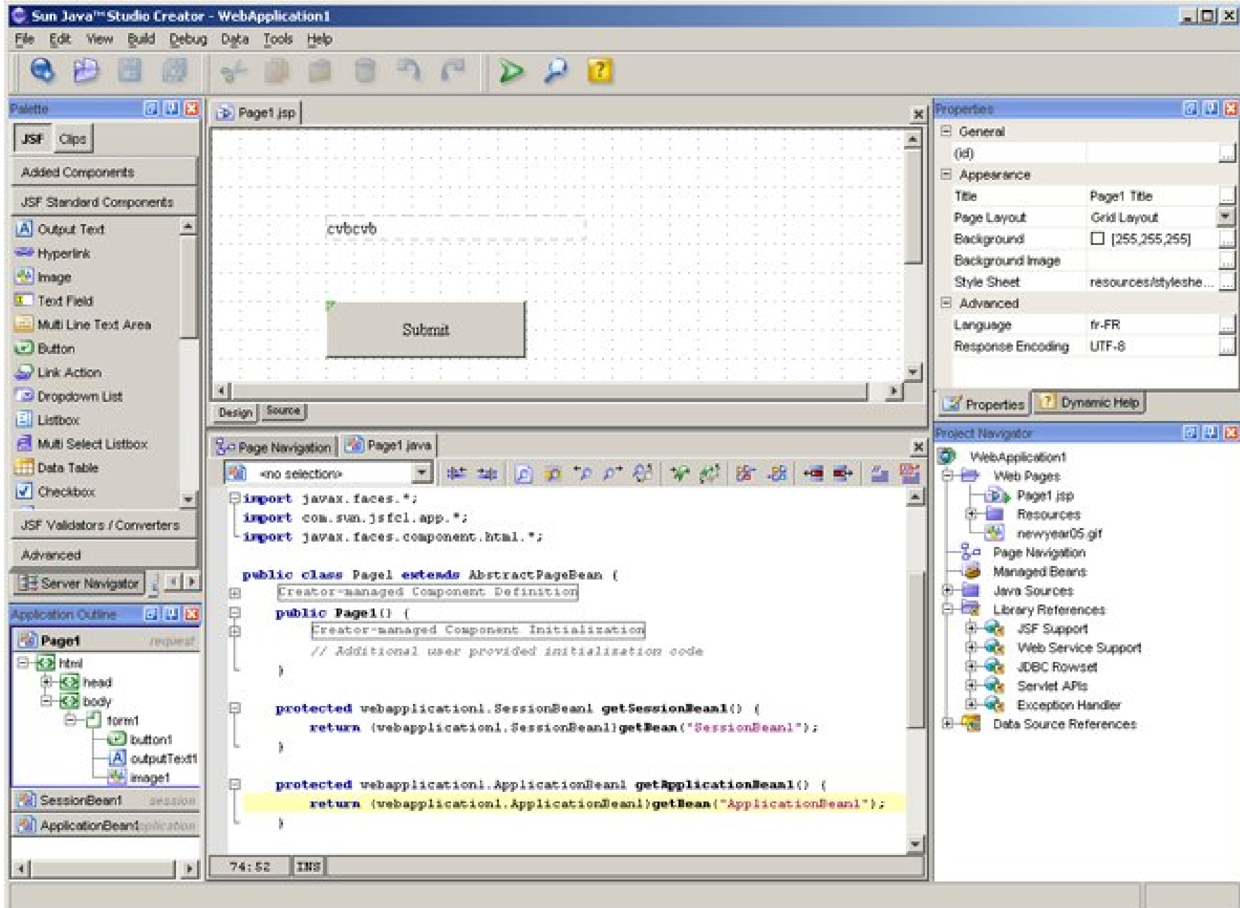
- Le support des dernières technologies Java
- Le développement wysiwyg d'applications web avec les JSF (plug-in Visual web Pack)

68.1.6. Sun Java Studio Creator

L'environnement de développement intégré Java Studio Creator de Sun permet de générer des applications Web à l'aide de la technologie Java notamment avec les Java ServerFaces et les portlets.

Sun Java Studio Creator version 1.0

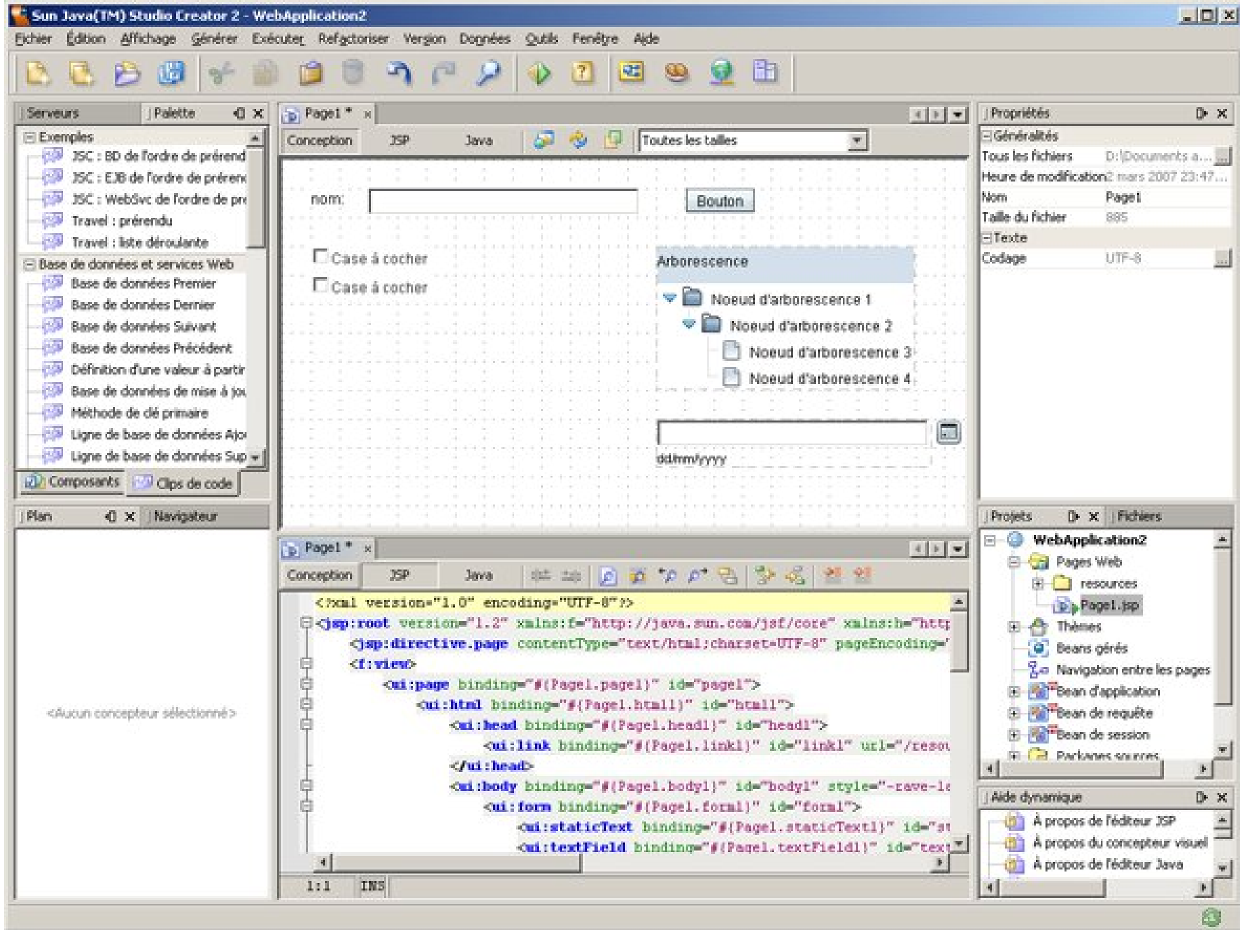
Cette première version de l'outil est payante. C'est un des premiers outil à exploiter les possibilités pour faciliter la mise en oeuvre des JSF dans un outil graphique.



Sun Java Studio Creator version 2.0

Cette seconde version est téléchargeable gratuitement après une inscription au SDN (Sun Developer Network).

Java Studio Creator 2 utilise Netbeans 4.1 comme base. Le package d'installation contient un JDK, le serveur d'application Sun Java System Application Server 8.x et une base de données



68.1.7. CodeGear (Borland) JBuilder

Borland est spécialisé depuis des années dans la création d'outils de développement possédant une excellente réputation. Ainsi Jbuilder est un IDE ergonomique qui génère un code "propre". Depuis sa version 3.5, JBuilder est écrit en Java ce qui lui permet de s'exécuter sans difficulté sur plusieurs plateformes notamment Windows, Linux ou Solaris.

Le produit dispose de nombreuses caractéristiques qui facilitent le travail du développeur : la technologie CodeInsight facilite grandement l'écriture du code dans l'éditeur, de nombreux assistants facilitent la génération de code ...

La version 2006 existe en plusieurs éditions :

- foundation : téléchargeable gratuitement
- developer
- entreprise

La version courante est la 2007 : elle est maintenue et diffusée par sa filiale CodeGear. Le site officiel du produit est à l'url www.codegear.com/Products/JBuilder/tabid/102/Default.aspx

Il existe plusieurs éditions :

- JBuilder Foundation 2007 (téléchargeable gratuitement après enregistrement)
- JBuilder Developper 2007
- JBuilder Professional 2007
- Jbuilder Enterprise 2007

La version 2007 est basée sur Eclipse.

68.1.8. JCreator

JCreator est un IDE développé par Xinox particulièrement rapide car il est écrit en code natif. Le site officiel du produit est à l'url www.jcreator.com.

La version courante est la 4.0.

Il existe deux éditions :

- La version LE : téléchargeable gratuitement
- La version Pro : payante

68.1.9. Oracle JDeveloper

Jdeveloper est un IDE riche en fonctionnalités qui couvre de nombreux aspect du cycle de vie du développement : modélisation UML, écriture du code, débogage, tests, profiling et déploiement d'applications.

Ecrit en Java, Jdeveloper est disponible sur plusieurs plateformes : Windows, Mac, Linux et plusieurs Unix.

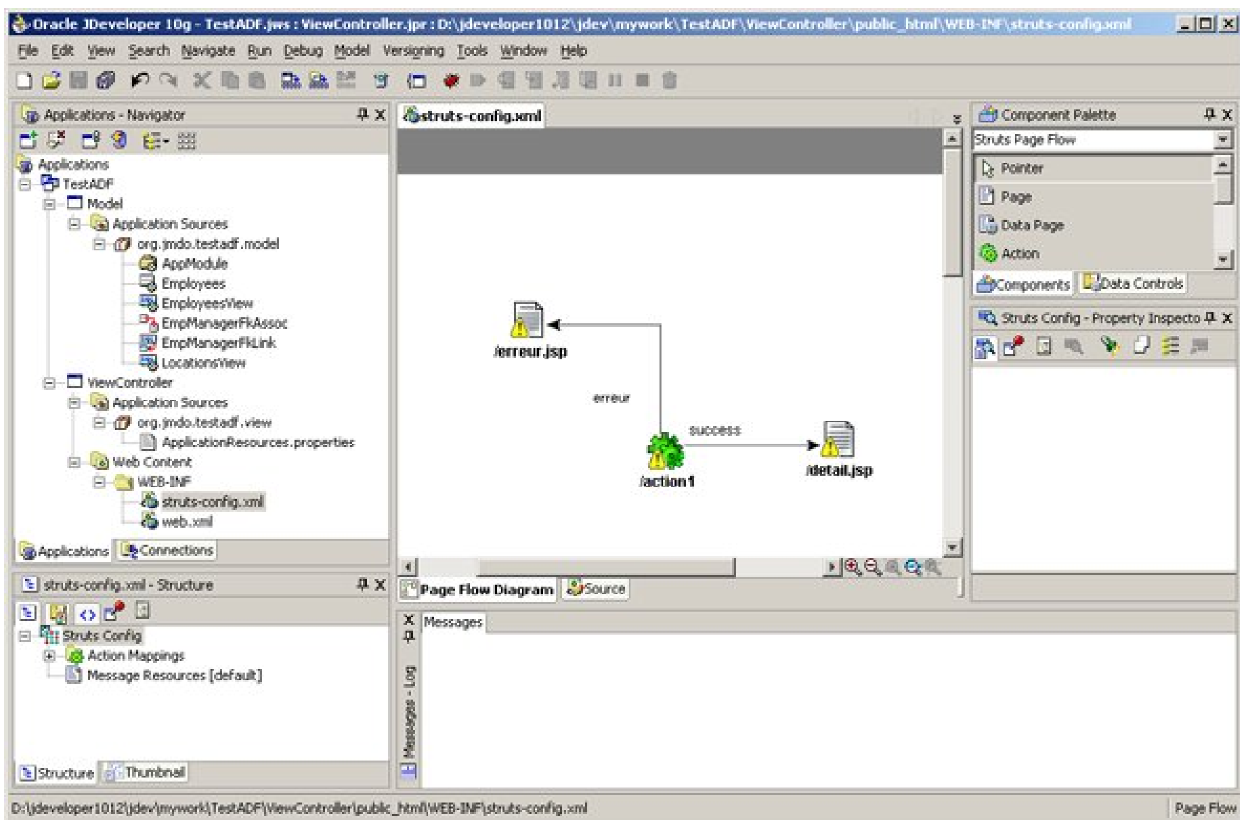
Le site officiel du produit est à l'url : www.oracle.com/technology/products/jdev/index.html

Jdeveloper propose des extensions pour enrichir l'outil en fonctionnalité notamment ceux proposés par des tiers.

Jdeveloper propose bien sûre une intégration facilitée de plusieurs produits d'Oracle notamment la base de données et le serveur d'application et surtout une forte intégration et une mise en oeuvre d'Oracle ADF.

Jdeveloper est disponible gratuitement après enregistrement chez OTN.

JDeveloper version 10.1.2



Cette version propose de nombreuses fonctionnalités dont voici quelques unes des plus intéressantes :

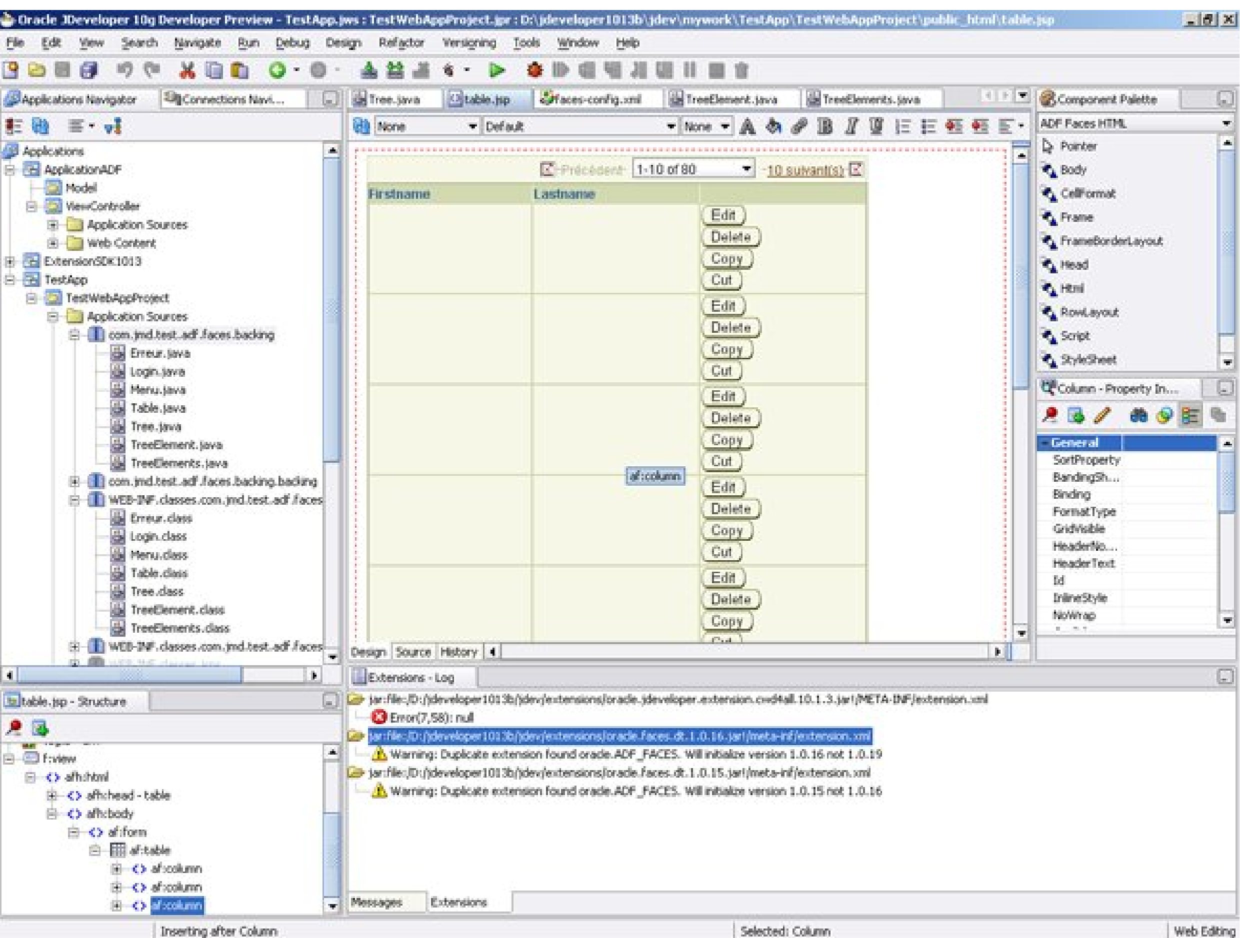
- Support de J2EE 1.4

- Un serveur d'application intégré à l'outil (OC4J)
- Support de nombreux outils open source (Ant, Junit, Struts, ...)
- Editeur de diagrammes pour les flux Struts
- Support de Toplink pour le mapping O/R
- Support d'ADF avec de nombreux assistant pour faciliter sa mise en oeuvre

JDeveloper version 10.1.3

La version 10.1.3.2 apporte de nombreuses fonctionnalités et améliorations par rapport à la version précédente dont voici quelques unes des principales :

- Une nouvelle interface plus moderne
- De nombreuses améliorations sont ajoutées dans les fonctionnalités de bases pour rattraper le retard de l'outil en la matière (assistant de code plus poussé, refactoring enrichi, historique local, ...)
- Support de nombreuses technologies (EJB 3.0, portlets, ADF, services web, XML, ...)
- Ajout de nouveaux éditeurs et designer : BEPL, ESB, XSLT
- Téléchargement des mises à jour à partir de l'outil



Jdeveloper 10.1.3 est proposé en trois versions :

- Studio : propose toutes les fonctionnalités dont ADF
- J2EE : propose de nombreuses fonctionnalités sauf ADF
- Java : permet le développement avec Java et XML

Jdeveloper est particulièrement intéressant pour mettre en oeuvre le framework Oracle ADF.

68.1.10. IntelliJ IDEA

IntelliJ est un IDE développé par JetBrains. Le site officiel du produit est à l'url www.jetbrains.com/idea/.

A partir de la version 9.0 publiée fin 2009, IntelliJ IDEA est proposé en deux éditions :

- Community Edition : cette édition open source sous licence Apache 2.0 propose un support pour Java SE et Groovy : assistance au codage, refactoring, debugging, support des tests (JUnit et TestNG), support de Maven et Ant
- Ultimate Edition : cette édition commerciale propose toutes les fonctionnalités de l'IDE notamment un support pour le développement d'application d'entreprises avec Java EE

http://www.jetbrains.org/

IntelliJ est disponible sous Windows, Linux et Mac OS X.

68.1.11. BEA Workshop

BEA Workshop est une famille d'IDE développée par BEA qui utilise Eclipse comme base. Le site officiel du produit est à l'url
www.bea.com/framework.jsp?CNT=index.htm&FP=/content/products/workshop/

Il existe plusieurs éditions :

- BEA Workshop for Weblogic
- BEA Workshop for JSP (cette édition est téléchargeable gratuitement après inscription)
- BEA Workshop Studio
- BEA Workshop for Struts
- BEA Workshop for JSF

68.1.12. IBM Visual Age for Java

IBM proposait une famille d'outils pour le développement avec différents langages dont une version dédiée à Java.

VAJ n'est plus supporté par IBM : il est remplacé par la famille d'outils Websphere Studio Application Developper.

Visual Age for Java (VAJ) était un outil novateur dans son ergonomie et son utilisation qui étaient complètement différentes des autres EDI. Les débuts de son utilisation étaient parfois déroutants mais une persévérance permettait de révéler toute sa puissance.

http://www-4.ibm.com/software/ad/vajava/

La fenêtre principale (plan de travail) est séparée en deux parties :

- l'espace de travail : il contient et organise les différents éléments (projets, packages, classes, méthodes ...)
- le code source : si l'élément sélectionné dans l'espace de travail contient du code, il est visualisé et modifiable dans cette partie

Par défaut le code est éditable par méthode mais depuis la version 3.5, il est toutefois possible de visualiser le code source complet mais les opérations réalisables dans ce mode sont moins nombreuses.

VAJ possédait plusieurs points forts : le regroupement de toutes les classes et leur organisation dans l'espace de travail, la compilation incrémentale à l'écriture et au débogage, le travail collaboratif avec le contrôle de version dans un référentiel (repository). Tous ces points facilitaient le développement de gros projets.

VAJ était un outil puissant particulièrement adapté aux utilisateurs chevronnés pour de gros projets.

68.1.13. Webgain Visual Café

Webgain Studio proposait un ensemble d'outils (Visual Café, Dreamweaver Ultradev, Top link, Structure Builder, Weblogic) pour la création d'applications e-business. Visual Café était l'IDE de développement en Java. Visual Café existait en trois versions : standard, expert et entreprise suite.

Malheureusement cet outil n'est plus disponible.

68.2. Les serveurs d'application

Les serveurs d'applications sont des outils qui permettent l'exécution de composants Java côté serveur (servlets, JSP, EJB, ...) selon les spécifications de la plate-forme J2EE/Java EE.

68.2.1. JBoss Application Server

JBoss est un projet open source développé en java pour fournir un serveur d'applications certifié J2EE 1.4.

Le site officiel de JBoss est à l'url www.jboss.org

JBoss est composé d'un ensemble d'outils : JBoss Server, JBoss MQ (implémentation de JMS), JBoss MX, JBoss TX (implémentation de JTA/JTS), JBoss SX , JBoss CX et JBoss CMP.

68.2.2. JOnAs

JOnAS est un projet open source développé par le consortium ObjectWeb (Bull, INRIA, Thales, France Telecom, Red Hat, Mandriva, ...) dont le but est de proposer un serveur d'application J2EE. JOnAs a obtenu la certification J2EE 1.4. Il se compose de nombreux éléments open source tel que JOTM pour le support des transactions, JORAM pour l'implémentation de JMS, Tomcat ou Jetty comme conteneur Web, Speedo pour l'implémentation JDO, ...

la page officiel de JOnAs est à l'url jonas.objectweb.org/index.html

68.2.3. GlassFish

Le projet GlassFish est un projet communautaire dont le but est de développer un serveur d'application open source qui implémente les spécifications de Java EE 5. Il doit devenir l'implémentation de référence de ces spécifications. Le site officiel du projet est à l'url GlassFish project web site. Un blog nommé The Aquarium permet d'obtenir des informations sur le projet GlassFish

Sun propose le serveur d'application Java System Application Server Platform Edition 9 qui implémente les spécifications de Java EE 5. Ce serveur gratuit est basé sur le projet GlassFish. Il peut être téléchargé dans une archive avec le SDK.

68.2.4. IBM Websphere Application Server

Websphere Application Server (WAS) est le serveur d'application de la famille d'outils Websphere. Il permet le déploiement de composants Java orienté entreprise.

http://www-306.ibm.com/software/webservers/appserv/was/

La version 4 est certifiée J2EE 1.2. Elle permet la mise en oeuvre des servlets, JSP, EJB et services Web (SOAP, UDDI, WSDL, XML). Cette version est proposée en 4 éditions qui supportent tout ou partie de ces composants :

- Standard Edition : pour les applications web utilisant des serlets, des JSP et XML
- Advanced Edition: supporte en plus les EJB, la répartition de charges sur plusieurs machines
- Advanced Single Server Edition : supporte toute les API J2EE mais uniquement sur une seule machine. Cette version ne peut pas être utilisée en production.
- Enterprise Edition : supporte en plus CORBA et la connexion aux ressources de l'entreprise

La version 5 est certifiée J2EE 1.3.

La version 6 est certifiée J2EE 1.4.

68.2.5. BEA Weblogic

Weblogic est une famille de produit proposé par BEA. Weblogic Server est un des leaders mondial des serveurs d'applications commerciaux.

http://fr.bea.com/produits/index.jsp

La version courante de cet outil est la 8.1.

68.2.6. Oracle Application Server

Oracle propose un serveur d'applications certifié J2EE 1.3.

La page officiel d'OAS est à l'url www.oracle.com/appserver/index.html

La version courante de cet outil est la 10g release 3.

68.2.7. Borland Entreprise Server

La page officiel du produit est à l'urlwww.borland.fr/besappserver/index.html

68.2.8. Macromedia JRun

JRun est l'implémentation d'un serveur d'applications de Macromedia.

http://www.macromedia.com/software/jrun/

La version 4 est certifié J2EE 1.3.

68.3. Les conteneurs web

Les conteneurs web sont des applications qui permettent d'exécuter du code Java utilisé pour définir des servlets et des JSP.

68.3.1. Apache Tomcat

Tomcat est un conteneur d'applications web (servlets et JSP) développé par la fondation Apache. C'est l'implémentation de référence pour les API servlets et JSP : il est donc pleinement compatible avec les spécifications J2EE de ces API.

http://jakarta.apache.org/tomcat/

L'utilisation de Tomcat est détaillée dans le chapitre «Tomcat».

68.3.2. Caucho Resin

Resin est un moteur de servlet et de JSP qui intègre un serveur web.

http://www.caucho.com/

68.3.3. Enhydra

Enhydra est un projet open source, initialement créé par Lutris technologies, pour développer un conteneur web pour Servlets et JSP. Il fourni en plus quelques fonctionnalités supplémentaires pour utiliser XML, mapper des données avec des objets et gérer un pool de connexion vers des bases de données

http://enhydra.enhydra.org/

68.4. Les conteneurs d'EJB

Les conteneurs d'EJB sont des applications qui fournissent un environnement d'exécution pour les EJB.

68.4.1. OpenEJB

OpenEJB est un projet open source pour développer un conteneur d'EJB qui respecte les spécifications 2.0 des EJB. La version 1.0 est distribuée en février 2006.

Le site du projet est l'url www.openejb.org/

68.5. Les outils divers

68.5.1. Jikes

Jikes est un compilateur Java open source écrit par IBM en code natif pour Windows et Linux. Son exécution est donc extrêmement rapide d'autant plus lorsqu'il s'agit de très gros projets sur une machine peu véloce.

La page de l'outil est à l'url www10.software.ibm.com/developerworks/opensource/jikes/

Pour utiliser Jikes, il suffit de décompresser l'archive et de mettre le fichier exécutable dans un répertoire inclus dans le CLASSPATH. Enfin, il faut déclarer une variable système JIKESPATH qui doit contenir les différents répertoires contenant les classes et les jars notamment le fichier rt.jar du JRE.

68.5.2. GNU Compiler for Java

GCJ fait parti du projet GCC (GNU Compiler Collection). Le projet GCC propose un compilateur pour plusieurs langages (C, C++, Objective C, Java ...) permettant de produire un exécutable pour plusieurs plateformes.

GCJ est donc un font-end pour utiliser GCC à partir de code Java. Il permet notamment de :

- de compiler du code source Java en byte-code
- de compiler du code source Java en un exécutable contenant du code machine dépendant d'un système d'exploitation

Pour un exécutable, le fichier final est lié avec une bibliothèque dédiée nommée libgcj qui contient entre autre les classes de bases et le ramasse miette.

La plupart des API de la plate-forme Java 2 sont supportées à l'exception notable de la bibliothèque AWT. Pour obtenir plus d'information sur la compatibilité, il suffit de consulter la page gcc.gnu.org/java/status.html

Son utilisation sous Windows nécessite un environnement particulier : CygWin ou MinGW (ce dernier étant retenu dans la suite de cette section).

Téléchargez sur le site www.mingw.org/download.shtml les fichiers : MinGW-3.1.0-1.exe (14,5 Mo) et MSYS-1.0.9.exe (2,7 Mo). (les noms de fichiers indiqués correspondent à la version courante au moment de l'écriture de cette section).

Lancez le programme MinGW-3.1.0-1.exe

Le programme d'installation se lance et demande une confirmation de l'installation : cliquer sur « Oui ». Un assistant permet de guider les différentes étapes de l'installation :

- Cliquez sur « Next »
- Lisez la licence et cliquez sur « Yes » si vous l'acceptez
- Lisez les informations et cliquez sur « Next »
- Choisissez le répertoire d'installation et cliquez sur « Next » (pour la suite des instructions, le répertoire par défaut c:\MinGW est utilisé)
- Cliquez sur « Install »
- Une fois l'installation terminée, cliquez sur « Finish »

Lancez le programme MSYS-1.0.9.exe

Le programme d'installation se lance et demande une confirmation de l'installation : cliquer sur « Oui ». Un assistant permet de guider les différentes étapes de l'installation :

* Cliquez sur « Next »
* Lisez la licence et cliquez sur « Yes » si vous l'acceptez
* Lisez les informations et cliquez sur « Next »
* Choisissez le répertoire d'installation et cliquez sur « Next » (pour la suite des instruction, le répertoire C:\MinGW\msys\1.0 est utilisé)
* Sélectionnez l'unique composant à installer et cliquez sur « Next »
* Sélectionnez le raccourci dans le menu Programme (MinGW par défaut) et cliquez sur « Next »
* Cliquez sur « Install »
* L'installation s'exécute et lance un script dos de configuration : il suffit de répondre aux questions

Exemple :

```
C:\MinGW\msys\1.0\postinstall>..\bin\sh.exe pi.sh
This is a post install process that will try to normalize between
your MinGW install if any as well as your previous MSYS installs
if any.  I don't have any traps as aborts will not hurt anything.
Do you wish to continue with the post install? [yn ] y
Do you have MinGW installed? [yn ] y
Please answer the following in the form of c:/foo/bar.
Where is your MinGW installation? c:/Mingw
Creating /etc/fstab with mingw mount bindings.
        Normalizing your MSYS environment.
You have script /bin/awk
You have script /bin/cmd
You have script /bin/echo
You have script /bin/egrep
You have script /bin/ex
You have script /bin/fgrep
You have script /bin/printf
You have script /bin/pwd
You have script /bin/rvi
You have script /bin/rview
You have script /bin/rvim
You have script /bin/vi
You have script /bin/view
Oh joy, you do not have c:/Mingw/bin/make.exe. Keep it that way.
C:\MinGW\msys\1.0\postinstall>pause
Appuyez sur une touche pour continuer...
```

* Appuyer sur une touche pour fermer la boîte dos
* Une fois l'installation terminée, cliquez sur « Finish »

Remarque : il est fortement recommandé de ne pas utiliser d'espace dans les noms des répertoires d'installation de MinGW et de MSYS.

Il faut pour plus de facilité d'utilisation ajouter à la variable PATH de l'environnement système les répertoires C:\MinGW\bin et C:\MinGW\msys\1.0\bin.

La version de GCC fournie avec MinGW précédemment installé est la 3.2. Pour utiliser GCJ, il faut utiliser la 3.3 et donc opérer une mise à jour.

Il faut télécharger les fichiers gcc-core-3.3.1-20030804-1.tar.gz, gcc-g++-3.3.1-20030804-1.tar.gz et gcc-java-3.3.1-20030804-1.tar.gz, les décompresser et extraire l'image tar dans le répertoire c:\MinGW.

Remarque : en standard aucun outil ne permet de traiter des fichiers gz et tar. Il faut utiliser un outil tiers.

Pour s'assurer de la bonne installation, il suffit d'ouvrir une boite Dos et d'exécuter la commande gcj. Le message suivant doit apparaître : gcj: no input files

Voici un petit exemple très simple de mise en oeuvre de GCJ.

Exemple du code à compiler :

```
public class Bonjour {
            public static void main(String[] args) {
```

```
                        System.out.println("Bonjour");
            }
}
```

Exemple de compilation et d'exécution :

```
D:\java\test\gcj>gcj -o Bonjour Bonjour.java -O --main=Bonjour
D:\java\test\gcj>dir
 Le volume dans le lecteur D s'appelle DATA
 Le numéro de série du volume est 34B2-159D
 Répertoire de D:\java\test\gcj
01/12/2003  15:19       <DIR>          .
01/12/2003  15:19       <DIR>          ..
01/12/2003  15:19            2 747 919 Bonjour.exe
01/12/2003  15:17                  108 Bonjour.java
01/12/2003  14:07                  141 Bonjour.java.bak
               4 fichier(s)        5 496 087 octets
               2 Rép(s)     560 402 432 octets libres
D:\java\test\gcj>bonjour
Bonjour
D:\java\test\gcj>
```

L'option -o permet de préciser le nom du fichier final généré.

L'option -main= permet de préciser la classe qui contient la méthode main() à lancer par l'exécutable.

GCJ peut être utilisé pour compiler le code source en byte-code grâce à l'option -C.

Exemple :

```
D:\java\test\gcj>gcj -C Bonjour.java
D:\java\test\gcj>dir
 Le volume dans le lecteur D s'appelle DATA
 Le numéro de série du volume est 34B2-159D
 Répertoire de D:\java\test\gcj
01/12/2003  15:29       <DIR>          .
01/12/2003  15:29       <DIR>          ..
01/12/2003  15:29                  389 Bonjour.class
01/12/2003  15:19            2 747 919 Bonjour.exe
01/12/2003  15:17                  108 Bonjour.java
               4 fichier(s)        2 748 557 octets
```

Le byte généré est légèrement plus compact que celui généré par la commande javac du jdk 1.4.1

Exemple :

```
D:\java\test\gcj>javac Bonjour.java
D:\java\test\gcj>dir
 Le volume dans le lecteur D s'appelle DATA
 Le numéro de série du volume est 34B2-159D
 Répertoire de D:\java\test\gcj
01/12/2003  15:29       <DIR>          .
01/12/2003  15:29       <DIR>          ..
01/12/2003  15:31                  405 Bonjour.class
01/12/2003  15:19            2 747 919 Bonjour.exe
01/12/2003  15:17                  108 Bonjour.java
               4 fichier(s)        2 748 573 octets
```

L'option -d permet de préciser un répertoire qui va contenir les fichiers .class généré par l'option -C.

68.5.3. Artistic Style

Artistic Style est un outil open source qui permet d'indenter et de formater un code source C, C++ et java

Le site du projet est à l'url http://sourceforge.net/projects/astyle/

Cet outil possède de nombreuses options de formatage de fichiers source. Les options les plus courantes pour un code source java sont :

astyle -jp --style=java nomDuFichier.java

Par défaut, l'outil conserve le fichier original en le suffixant par .orig.

68.6. Les MOM

Les Middleware Oriented Message sont des outils qui permettent l'échange de messages entre des composants d'une application ou entre applications. Pour pouvoir les utiliser avec Java, ils doivent implémenter l'API JMS de Sun.

68.6.1. OpenJMS

OpenJMS est une implémentation Open Source des spécifications JMS

Le site officiel du projet est à l'url http://openjms.sourceforge.net/

Pour utiliser OpenJMS, il faut télécharger l'archive qui contient OpenJMS : par exemple fichier openjms-0.7.7-beta-1.zip

Pour installer OpenJMS, il suffit de décompresser le fichier zip téléchargé dans un répertoire du système.

Exemple :

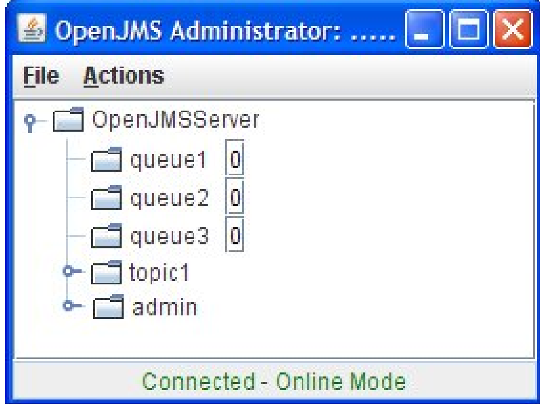
```
C:\java>jar xvf openjms-0.7.7-beta-1.zip
```

La décompression créé un répertoire nommé openjms-0.7.7-beta-1

Pour démarrer et arrêter le serveur, il faut utiliser respectivement les scripts startup et shutdown du sous répertoire bin du répertoire d'installation.

Pour lancer la console d'administration, il faut utiliser le script admin

Cliquez sur « Actions / Connections / OnLine »



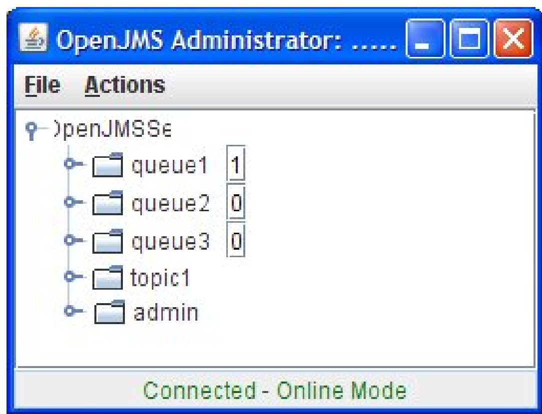
Il est alors possible d'écrire une application qui utiliser JMS et les queues, par exemple l'application TestOpenJMS1 fournie dans le chapitre sur «JMS (Java Messaging Service)».

Les paramètres JNDI peuvent être fournis dans un fichier de configuration nommé jndi.properties

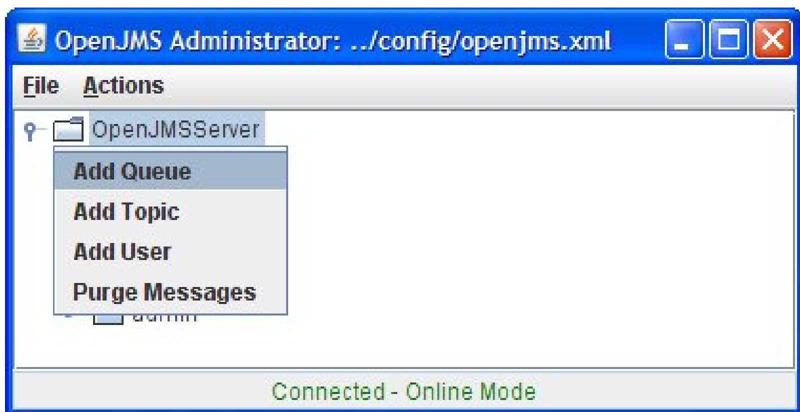
Exemple : jndi.properties

```
java.naming.provider.url=tcp://localhost:3035
java.naming.factory.initial=org.exolab.jms.jndi.InitialContextFactory
java.naming.security.principal=admin
java.naming.security.credentials=openjms
```

Dans la console d'administration, cliquez sur « Actions / Refresh »



La configuration des queues est stockée dans le fichier openjms.xml. La console d'administration permet de gérer les destinations (ajout, suppression, ...)

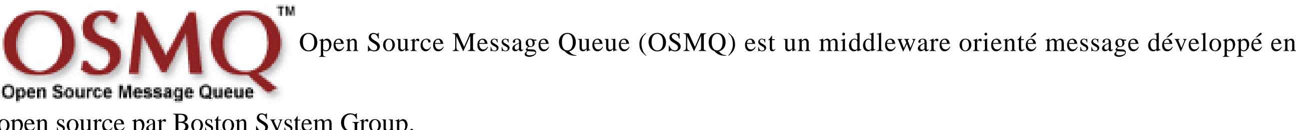


68.6.2. Joram

Joram est l'acronyme de Java Open Reliable Asynchronous Messaging. C'est une implémentation open source des spécifications JMS 1.1.

La page officiel de cet outil est à l'url http://www.objectweb.org/joram/

68.6.3. OSMQ



Open Source Message Queue (OSMQ) est un middleware orienté message développé en open source par Boston System Group.

Le site de ce produit est à l'url http://www.osmq.org/

68.7. Les outils pour bases de données

68.7.1. Derby

Derby est un SGBDR open source écrit en Java et maintenu le projet Apache.

Historiquement, c'est un produit développé par Cloudscape acquis par IBM (lors de son rachat d'Informix) qui en a fait dont à la fondation Apache.

Derby est aussi intégré dans des produits de Sun notamment le JDK 6.0 et GlassFish sous le nom Java DB

Le site officiel est à l'url http://db.apache.org/derby/

68.7.2. SQuirrel-SQL

Squirrel-SQL est un client SQL open source écrit en Java. Il permet au travers d'une interface graphique de consulter et de manipuler une base de données pourvue d'un pilote JDBC.

Le site de l'outil est à l'url : http://squirrel-sql.sourceforge.net/

L'éditeur SQL propose une complétion de code (nom de table, de colonnes, ...).

Les données sont éditables dans l'interface graphique.

SQuirrel est extensible au travers de plug-in dont plusieurs sont fournis par défaut.

Pour installer Squirrel il faut télécharger le fichier squirrel-sql-<version>-install.jar qui est un setup d'installation au format IzPack.

Pour l'exécuter l'installation il faut exécuter :

```
java -jar squirrel-sql-<version>-install.jar
```

68.8. Les outils de modélisation UML

68.8.1. Argo UML

Argo UML est un projet open source écrit en java qui vise à développer un outil de modélisation UML 1.1. Il est possible de créer des diagrammes UML et de générer le code Java correspondant aux diagrammes de classes. Une option permet de créer les diagrammes de classes à partir du code source java.

http://argouml.tigris.org/

Cet outil n'est pas encore en version finale mais la version 0.9.5 offre de nombreuses fonctionnalités.

68.8.2. Poseidon for UML

http://gentleware.com/index.php

Plusieurs éditions de Poseidon for UML sont proposées dont la version community edition qui est disponible gratuitement.

La version 4.1 propose de nombreuses fonctionnalités dont le respect de la version UML 2.0

68.8.3. StarUML

StarUML est historiquement un produit commercial nommé Plastic puis Agora Plastic qui est devenu un projet open source (licence GPL) en 2005 renommé en StarUML.

Le site officiel de StarUML est à l'url http://www.staruml.com/

Il ne fonctionne que sous Windows mais la version 5.0 propose des fonctionnalités intéressantes :

- support de UML 2.0 et de MDA
- le support des designs patterns (GoF et EJB)
- importation des fichiers Rational Rose
- exportation en XMI
- génération de code et retro conception (Java, C#, C++)
- extensible par plug-in reposant sur la technologie COM

69. Ant

Ant est un projet du groupe Apache-Jakarta. Son but est de fournir un outil écrit en Java pour permettre la construction d'applications (compilation, exécution de tâches post et pré compilation, ...). Ces processus de construction d'applications sont très importants car ils permettent d'automatiser des opérations répétitives tout au long du cycle de développement de l'application (développement, tests, recettes, mises en production, ...). Le site officiel de l'outil Ant est http://jakarta.apache.org/ant/index.html.

Ant pourrait être comparé au célèbre outil make sous Unix. Il a été développé pour fournir un outil de construction indépendant de toute plate-forme. Ceci est particulièrement utile pour des projets développés sur et pour plusieurs systèmes ou pour migrer des projets d'un système sur un autre. Il est aussi très efficace pour de petits développements.

Ant repose sur un fichier de configuration XML qui décrit les différentes tâches qui devront être exécutées par l'outil. Ant fournit un certain nombre de tâches courantes qui sont codées sous forme d'objets développés en Java. Ces tâches sont donc indépendantes du système sur lequel elles seront exécutées. De plus, il est possible d'ajouter ces propres tâches en écrivant de nouveaux objets Java respectant certaines spécifications.

Le fichier de configuration contient un ensemble de cibles (target). Chaque cible contient une ou plusieurs tâches. Chaque cible peut avoir une dépendance envers une ou plusieurs autres cibles pour pouvoir être exécutée.

Fichier de configuration Ant
Cible
Tache
Cible
Tache
Tache

Les environnements de développement intégrés proposent souvent un outil de construction propriétaire qui son généralement moins souple et moins puissant que Ant. Ainsi des plug-ins ont été développés pour la majorité d'entre eux (JBuilder, Forte, Visual Age, ...) pour leur permettre d'utiliser Ant, devenu un standard de fait.

Ant possède donc plusieurs atouts : multi plate-forme, configurable grâce à un fichier XML, open source et extensible.

Pour obtenir plus de détails sur l'utilisation de l'outil Ant, il est possible de consulter la documentation de la version courante à l'url suivante : http://jakarta.apache.org/ant/manual/index.html

Une version 2 de l'outil Ant est en cours de développement.

Ce chapitre contient plusieurs sections :

- L'installation de l'outil Ant
- Exécuter ant
- Le fichier build.xml
- Les tâches (task)

69.1. L'installation de l'outil Ant

Pour pouvoir utiliser Ant, il faut avoir un JDK 1.1 ou supérieur et installer Ant sur la machine.

69.1.1. L'installation sous Windows

Le plus simple est de télécharger la distribution binaire de l'outil Ant pour Windows : jakarta-ant-version-bin.zip sur le site de Ant.

Il suffit ensuite de :

- décompresser le fichier (un répertoire jakarta-ant-version est créer, contenant l'outil et sa documentation)
- ajouter le chemin complet au répertoire bin de l'outil Ant à la variable système PATH (pour pouvoir facilement appeler Ant n'importe où dans l'arborescence du système)
- s'assurer que la variable JAVA_HOME pointe sur le répertoire contenant le JDK
- créer une variable d'environnement ANT_HOME qui pointe sur le répertoire jakarta-ant-version créé lors de la décompression du fichier
- il peut être nécessaire d'ajouter les fichiers .jar contenus dans le répertoire lib de l'outil Ant à la variable d'environnement CLASSPATH

Exemple de lignes contenues dans le fichier autoexec.bat :

```
...
set JAVA_HOME=c:\jdk1.3 set
ANT_HOME=c:\java\ant
set PATH=%PATH%;%ANT_HOME%\bin
...
```

69.2. Exécuter ant

Ant s'utilise en ligne de commande avec la syntaxe suivante :

ant [options] [cible]

Par défaut, Ant recherche un fichier nommé build.xml dans le répertoire courant. Ant va alors exécuter la cible par défaut définie dans le projet de ce fichier build.xml.

Il est possible de préciser le nom du fichier de configuration en utilisant l'option -buildfile et en la faisant suivre du nom du fichier de configuration.

Exemple :

```
ant -buildfile monbuild.xml
```

Il est possible de préciser une cible précise à exécuter. Dans ce cas, Ant exécute les cibles dont dépend la cible précisée et exécute cette dernière.

Exemple : exécuter la cible clean et toutes les cibles dont elle dépend

```
ant clean
```

Ant possède plusieurs options dont voici les principales :

Option	Rôle
-quiet	fourni un minimum d'informations lors de l'exécution
-verbose	fourni un maximum d'informations lors de l'exécution
-version	affiche la version de l'outil ant
-projecthelp	affiche les cibles définis avec leur description
-buildfile	permet de préciser le nom du fichier de configuration
-Dnom=valeur	permet de définir une propriété dont le nom et la valeur sont séparés par un caractère =

69.3. Le fichier build.xml

Le fichier build est un fichier XML qui contient la description du processus de construction de l'application.

Comme tout document XML, le fichier débute par un prologue :

<?xml version="1.0">

L'élément principal de l'arborescence du document est le projet représenté par le tag <project>>project< qui est donc le tag racine du document.

A l'intérieur du projet, il faut définir les éléments qui le composent :

- les cibles (targets) : ce sont des étapes du projet de construction
- les propriétés (properties) : ce sont des variables qui contiennent des valeurs utilisables par d'autres éléments (cibles ou tâches)
- les tâches (tasks) : ce sont des traitements unitaires à réaliser dans une cible donnée

Pour permettre l'exécution sur plusieurs plateformes, les chemins de fichiers utilisés dans le fichier build.xml doivent utiliser le caractère slash '/' comme séparateur même sous Windows qui utilise le caractère anti-slash '\'.

69.3.1. Le projet

Il est défini par le tag racine <project> dans le fichier build.

Ce tag possède plusieurs attributs :

- name : cet attribut précise le nom du projet
- default : cet attribut précise la cible par défaut à exécuter si aucune cible n'est précisée lors de l'exécution
- basedir : cet attribut précise le répertoire qui servira de référence pour l'utilisation de localisation relative des autres répertoires.

Exemple :

```
  <project name="mon projet" default="compile" basedir=".">
```

69.3.2. Les commentaires

Les commentaires sont inclus dans un tag <!-- -->.

Exemple :

```
<!-- Exemple de commentaires -->
```

69.3.3. Les propriétés

Le tag <property> permet de définir une propriété qui pourra être utilisée dans le projet : c'est souvent la définition d'un répertoire ou d'une variable qui sera utilisée par certaines tâches. Leur définition en tant que propriété permet de facilement changer leur valeur une seule fois même si la valeur de la propriété est utilisée plusieurs fois dans le projet.

Exemple :

```
<property name= "nom_appli" value= "monAppli"/>
```

Les propriétés sont immuables et peuvent être définies de deux manières :

- avec le tag <property>
- avec l'option -D sur la ligne de commande lors de l'appel de la commande ant

Pour utiliser une propriété sur la ligne de commande, il faut utiliser l'option -D immédiatement suivi du nom de la propriété, suivi du caractère =, suivi de la valeur, le tout sans espace.

Le tag <property> possède plusieurs attributs :

- name : cet attribut défini le nom de la propriété
- value : cet attribut défini la valeur de la propriété
- location : cet attribut permet de définir un fichier avec son chemin absolu. Il peut être utilisé à la place de l'attribut value
- file : cet attribut permet de préciser le nom d'un fichier qui contient la définition d'un ensemble de propriétés. Ce fichier sera lu et les propriétés qu'il contient seront définies.

L'utilisation de l'attribut file est particulièrement utile car il permet de séparer la définition des propriétés du fichier build. Le changement d'un paramètre ne nécessite alors pas de modifications dans le fichier xml build.

Exemple :

```
<property file="mesproprietes.properties" />
<property name="repSources" value="src" />
<property name="projet.nom" value="mon_projet" />
<property name="projet.version" value="0.0.10" />
```

L'ordre de définition d'une propriété est très important : Ant gère une priorité sur l'ordre de définition d'une propriété. La règle est la suivante : la première définition d'une propriété est prise en compte, les suivantes sont ignorées.

Ainsi, les propriétés définies via la ligne de commande sont prioritaires par rapport à celles définies dans le fichier build. Il est aussi préférable de mettre le tag <property> contenant un attribut file avant les tags <property> définissant des variables.

Pour utiliser une propriété définie dans le fichier, il faut utiliser la syntaxe suivante :
${nom_propriete}

Exemple :
${repSources}

Il existe aussi des propriétés prédéfinies par Ant et utilisables dans chaque fichier build :

Propriété	Rôle
basedir	chemin absolu du répertoire de travail (cette valeur est précisée dans l'attribut basedir du tag project)
ant.file	chemin absolu du fichier build en cours de traitement
ant.java.version	version de la JVM qui exécute ant
ant.project.name	nom du projet en cours d'utilisation

69.3.4. Les ensembles de fichiers

Le tag <fileset> permet de définir un ensemble de fichiers. Cet ensemble de fichier sera utilisé dans une autre tâche. La définition d'un tel ensemble est réalisée grâce à des attributs du tag <fileset> :

Attribut	Rôle
dir	Définit le répertoire de départ de l'ensemble de fichiers
includes	Liste des fichiers à inclure
excludes	Liste des fichiers à exclure

L'expression **/ permet de désigner tous les sous répertoires du répertoire défini dans l'attribut dir.

Exemple :

```
<fileset dir="src" includes="**/*.java">
```

69.3.5. Les ensembles de motifs

Le tag <patternset> permet de définir un ensemble de motifs pour sélectionner des fichiers.

La définition d'un tel ensemble est réalisée grâce à des attributs du tag <patternset> :

Attribut	Rôle
id	Définit un identifiant pour l'ensemble qui pourra ainsi être réutilisé
includes	Liste des fichiers à inclure
excludes	Liste des fichiers à exclure
refid	Demande la réutilisation d'un ensemble dont l'identifiant est fourni comme valeur

L'expression **/ permet de désigner tous les sous répertoires du répertoire définit dans l'attribut dir. Le caractère ? représente un unique caractère quelconque et le caractère * représente zéro ou n caractères quelconques.

Exemple :

```
<fileset dir="src">
```

```
  <patternset id="source_code">
    <includes="**/*.java"/>
  </patternset>
</fileset>
```

69.3.6. Les listes de fichiers

Le tag <filelist> permet de définir une liste de fichiers finis. Chaque fichier est nommément ajouté dans la liste, séparé chacun par une virgule. La définition d'un tel élément est réalisée grâce à des attributs du tag <filelist> :

Attribut	Rôle
id	Définit un identifiant pour la liste qui pourra ainsi être réutilisé
dir	Définit le répertoire de départ de la liste de fichiers
files	liste des fichiers séparés par une virgule
refid	Demande la réutilisation d'une liste dont l'identifiant est fourni comme valeur

Exemple :

```
<filelist dir="texte" files="fichier1.txt,fichier2.txt" />
```

69.3.7. Les éléments de chemins

Le tag <pathelement> permet de définir un élément qui sera ajouté à la variable classpath. La définition d'un tel élément est réalisée grâce à des attributs du tag <pathelement> :

Attribut	Rôle
location	Défini un chemin d'une ressource qui sera ajoutée
path	

Exemple :

```
<classpath>
  <pathelement location="bin/mabib.jar">
  <pathelement location="lib/">
</classpath>
```

Il est préférable pour assurer une meilleure compatibilité entre plusieurs systèmes d'utiliser des chemins relatifs par rapport au répertoire de base de projet.

69.3.8. Les cibles

Le tag <target> définit une cible. Une cible est un ensemble de tâches à réaliser dans un ordre précis. Cet ordre correspond à celui des tâches décrites dans la cible.

Le tag >target< possède plusieurs attributs :

- name : contient le nom de la cible. Cet attribut est obligatoire

* description : contient une brève description de la cible. Cet attribut est optionnel mais il est recommandé de l'utiliser car la plupart des IDE l'affiche lors de l'utilisation de l'outil ant
* if : permet de conditionner l'exécution par l'existence d'une propriété. Cet attribut est optionnel
* unless : permet de conditionner l'exécution par l'inexistence de la définition d'une propriété. Cet attribut est optionnel
* depends : permet de définir la liste des cibles dont dépend la cible. Cet attribut est optionnel

Il est possible de faire dépendre une cible d'une ou plusieurs autres cibles du projet. Lorsqu'une cible doit être exécutée, Ant s'assure que les cibles dont elle dépend ont été complètement exécutées préalablement depuis l'exécution de l'outil Ant. Une dépendance est définie grâce à l'attribut depends. Plusieurs cibles dépendantes peuvent être listées dans l'attribut depends. Dans ce cas, chaque cible doit être séparée avec une virgule.

69.4. Les tâches (task)

Une tâche est une unité de traitements contenue dans une classe Java qui implémente l'interface org.apache.ant.Task. Dans le fichier de configuration, une tâche est un tag qui peut avoir des paramètres pour configurer le traitement à réaliser. Une tâche est obligatoirement incluse dans une cible.

Ant fournit en standard un certain nombre de tâches pour des traitements courants lors du développement en Java :

Catégorie	Nom de la tâche	Rôle
Tâches internes	echo	Afficher un message
	dependset	Définir des dépendances entre fichiers
	taskdef	Définir une tâche externe
	typedef	Définir un nouveau type de données
Gestion des propriétés	available	Définir une propriété si une ressource existe
	condition	Définir une propriété si une condition est vérifiée
	pathconvert	Définir une propriété avec la conversion d'un chemin de fichier spécifique à un OS
	property	Définir une propriété
	tstamp	Initialiser les propriétés DSTAMP, TSTAMP et TODAY avec la date et heure courante
	uptodate	Définir une propriété en comparant la date de modification de fichiers
tâches Java	java	Exécuter une application dans la JVM
	javac	Compiler des sources Java
	javadoc	Générer la documentation du code source
	rmic	Générer les classes stub et skeleton nécessaires à la technologie rmi
	signjar	Signer un fichier jar
Gestion des archives	ear	Créer une archive contenant une application J2EE
	gunzip	Décompresser une archive
	gzip	Compresser dans une archive
	jar	Créer une archive de type jar
	tar	Créer une archive de type tar
	unjar	Décompresser une archive de type jar
	untar	Décompresser une archive de type tar

	unwar	Décompresser une archive de type war
	unzip	Décompresser une archive de type zip
	war	Créer une archive de type war
	zip	Créer une archive de type zip
tâches diverses	apply	Exécuter une commande externe appliquée à un ensemble de fichiers
	cvs	Gérer les sources dans CVS
	cvspass	
	exec	Exécuter une commande externe
	genkey	Générer une clé dans un trousseau de clé
	get	Obtenir une ressource à partir d'une URL
	mail	Envoyer un courrier électronique
	replace	Remplacer une chaîne de caractères par une autre
	sql	Exécuter une requête SQL
	style	Appliquer une feuille de style XSLT à un fichier XML
Gestion des fichiers	chmod	Modifier les droits d'un fichier
	copy	Copier un fichier
	delete	Supprimer un fichier
	mkdir	Créer un répertoire
	move	Déplacer ou renommer un fichier
	touch	Modifier la date de modification du fichier avec la date courante
Gestion de l'exécution de l'outil Ant	ant	Exécuter un autre fichier de build
	antcall	Exécuter une cible
	fail	Stopper l'exécution de l'outil Ant
	parallel	Exécuter une tâche en parallèle
	record	Enregistrer les traitements de l'exécution dans un fichier journal
	sequential	Exécuter une tâche en séquentielle
	sleep	Faire une pause dans les traitements

Certaines de ces tâches seront détaillées dans les sections suivantes : pour une référence complète de ces tâches, il est nécessaire de consulter la documentation de l'outil Ant.

69.4.1. echo

La tâche <echo> permet d'écrire dans un fichier ou d'afficher un message ou des informations durant l'exécution des traitements.

Les données à utiliser peuvent être fournies dans un attribut dédié ou dans le corps du tag <echo>.

Cette tâche possède plusieurs attributs dont les principaux sont :

Attribut	Rôle

message	Message à afficher
file	Fichier dans lequel le message sera inséré
append	Booléen qui précise si le message est ajouté à la fin du fichier (true) ou si le fichier doit être écrasé avec le message fourni (false)

Exemple :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<project name="Test avec Ant" default="init" basedir=".">
  <!-- ================================================================== -->
  <!-- Initialisation -->
  <!-- ================================================================== -->
  <target name="init">
    <echo message="Debut des traitements" />
    <echo>
      Fin des traitements du projet ${ant.project.name}
    </echo>
    <echo file="${basedir}/log.txt" append="false" message="Debut des traitements" />
    <echo file="${basedir}/log.txt" append="true" >
Fin des traitements
    </echo>
  </target>
</project>
```

Résultat :

```
C:\java\test\testant>ant
Buildfile: build.xml

init:
     [echo] Debut des traitements
     [echo]
     [echo]       Fin des traitements du projet Test avec Ant
     [echo]

BUILD SUCCESSFUL
Total time: 2 seconds
C:\java\test\testant>type log.txt
Debut des traitements
Fin des traitements

C:\java\test\testant>
```

69.4.2. mkdir

La tâche <mkdir> permet de créer un répertoire avec éventuellement ses répertoires pères ci ceux ci n'existent pas.

Cette tâche possède un seul attribut:

Attribut	Rôle
dir	Précise le chemin et le nom du répertoire à créer

Exemple :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<project name="Test avec Ant" default="init" basedir=".">
  <!-- ================================================================== -->
  <!-- Initialisation -->
  <!-- ================================================================== -->
  <target name="init">
    <mkdir dir="${basedir}/gen" />
  </target>
```

```
</project>
```

Résultat :

```
C:\java\test\testant>ant
Buildfile: build.xml

init:
    [mkdir] Created dir: C:\java\test\testant\gen

BUILD SUCCESSFUL
Total time: 2 seconds
C:\java\test\testant>
```

69.4.3. delete

La tâche <delete> permet de supprimer des fichiers ou des répertoires.

Cette tâche possède plusieurs attributs dont les principaux sont :

Attribut	Rôle
file	Permet de préciser le fichier à supprimer
dir	Permet de préciser le répertoire à supprimer
verbose	Booléen qui permet d'afficher la liste des éléments supprimés
quiet	Booléen qui permet de ne pas afficher les messages d'erreurs
includeEmptyDirs	Booléen qui permet de supprimer les répertoires vides

Exemple :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<project name="Test avec Ant" default="init" basedir=".">
  <!-- ============================================================== -->
  <!-- Initialisation -->
  <!-- ============================================================== -->
  <target name="init">
    <delete dir="${basedir}/gen" />
    <delete file="${basedir}/log.txt" />
    <delete>
      <fileset dir="${basedir}/bin" includes="**/*.class" />
    </delete>
  </target>
</project>
```

Résultat :

```
C:\java\test\testant>ant
Buildfile: build.xml

init:
   [delete] Deleting directory C:\java\test\testant\gen
   [delete] Deleting: C:\java\test\testant\log.txt

BUILD SUCCESSFUL
Total time: 2 seconds
C:\java\test\testant>
```

69.4.4. copy

La tâche <copy> permet de copier un ou plusieurs fichiers dans le cas ou ils n'existent pas dans la cible ou si ils sont plus récents dans la cible.

Cette tâche possède plusieurs attributs dont les principaux sont :

Attribut	Rôle
file	Désigne le fichier à copier
todir	Permet de préciser le répertoire cible dans lequel les fichiers seront copiés
overwrite	Booléen qui permet d'écraser les fichiers cibles si ils sont plus récents (false par défaut)

L'ensemble des fichiers concernés par la copie doit être précisé avec un tag fils <fileset>.

Exemple :

```
<project name="utilisation de hbm2java" default="init" basedir=".">

  <!-- Definition des proprietes du projet -->
  <property name="projet.sources.dir"    value="src"/>
  <property name="projet.bin.dir"        value="bin"/>

  <!-- Initialisation des traitements -->
  <target name="init" description="Initialisation">
    <!-- Copie des fichiers de mapping et parametrage -->
    <copy todir="${projet.bin.dir}" >
      <fileset dir="${projet.sources.dir}" >
        <include name="**/*.properties"/>
        <include name="**/*.hbm.xml"/>
        <include name="**/*.cfg.xml"/>
      </fileset>
    </copy>
  </target>
</project>
```

Résultat :

```
C:\java\test\testhibernate>ant
Buildfile: build.xml

init:
     [copy] Copying 3 files to C:\java\test\testhibernate\bin

BUILD SUCCESSFUL
Total time: 3 seconds
```

69.4.5. tstamp

La tâche <tstamp> permet de définir trois propriétés :

- DSTAMP : cette propriété est initialisée avec la date du jour au format AAAMMJJ
- TSTAMP : cette propriété est initialisée avec l'heure actuelle sous la forme HHMM
- TODAY : cette propriété est initialisée avec la date du jour au format long

Cette tâche ne possède pas d'attributs.

Exemple :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

```
<project name="Test avec Ant" default="init" basedir=".">
  <!-- ================================================================= -->
  <!-- Initialisation -->
  <!-- ================================================================= -->
  <target name="init">
    <tstamp/>
    <echo message="Nous sommes le ${TODAY}" />
    <echo message="DSTAMP = ${DSTAMP}" />
    <echo message="TSTAMP = ${TSTAMP}" />
  </target>
</project>
```

Résultat :

```
C:\java\test\testant>ant
Buildfile: build.xml

init:
     [echo] Nous sommes le August 25 2004
     [echo] DSTAMP = 20040825
     [echo] TSTAMP = 1413

BUILD SUCCESSFUL
Total time: 2 seconds
```

69.4.6. java

La tâche <java> permet de lancer une machine virtuelle pour exécuter une application compilée.

Cette tâche possède plusieurs attributs dont les principaux sont :

Attribut	Rôle
classname	nom pleinement qualifié de la classe à exécuter
jar	nom du fichier de l'application à exécuter
classpath	classpath pour l'exécution. Il est aussi possible d'utiliser un tag fils <classpath> pour le spécifier
classpathref	utilisation d'un classpath précédemment défini dans le fichier de build
fork	lancer l'exécution dans une JVM dédiée au lieu de celle ou l'exécute Ant
output	enregistrer les sorties de la console dans un fichier

Le tag fils <arg> permet de fournir des paramètres à l'exécution.

Le tag fils <classpath> permet de définir le classpath à utiliser lors de l'exécution

Exemple :

```
<project name="testhibernate1" default="TestHibernate1" basedir=".">

  <!-- Definition des proprietes du projet -->
  <property name="projet.sources.dir"    value="src"/>
  <property name="projet.bin.dir"        value="bin"/>
  <property name="projet.lib.dir"        value="lib"/>

  <!-- Definition du classpath du projet -->
  <path id="projet.classpath">
    <fileset dir="${projet.lib.dir}">
      <include name="*.jar"/>
    </fileset>
    <pathelement location="${projet.bin.dir}" />
```

```
  </path>

  <!-- Execution de TestHibernate1 -->
  <target name="TestHibernate1" description="Execution de TestHibernate1" >
    <java classname="TestHibernate1" fork="true">
      <classpath refid="projet.classpath"/>
    </java>
  </target>
</project>
```

69.4.7. javac

La tâche <javac> permet la compilation de fichiers source contenus dans une arborescence de répertoires.

Cette tâche possède plusieurs attributs dont les principaux sont :

Attribut	Rôle
srcdir	précise le répertoire racine de l'arborescence du répertoire contenant les sources
destdir	précise le répertoire où les résultats des compilations seront stockés
classpath	classpath pour l'exécution. Il est aussi possible d'utiliser un tag fils <classpath> pour le spécifier
classpathref	utilisation d'un classpath précédemment défini dans le fichier de build
nowarn	précise si les avertissements du compilateur doivent être affichés. La valeur par défaut est off
debug	précise si le compilateur doit inclure les informations de débogage dans les fichiers compilés. La valeur par défaut est off
optimize	précise si le compilateur doit optimiser le code compilé qui sera généré. La valeur par défaut est off
deprecation	précise si les avertissements du compilateur concernant l'usage d'éléments deprecated doivent être affichés. La valeur par défaut est off
target	précise la version de la plate-forme Java cible (1.1, 1.2, 1.3, 1.4, ...)
fork	lance la compilation dans une JVM dédiée au lieu de celle ou s'exécute Ant. La valeur par défaut est false
failonerror	précise si les erreurs de compilations interrompent l'exécution du fichier de build. La valeur par défaut est true
source	version des sources java : particulièrement utile pour Java 1.4 et 1.5 qui apportent des modifications à la grammaire du langage Java

Exemple :

```
<project name="compiltation des classes" default="compile" basedir=".">
  <!-- Definition des proprietes du projet -->
  <property name="projet.sources.dir" value="src"/>
  <property name="projet.bin.dir" value="bin"/>
  <property name="projet.lib.dir" value="lib"/>

  <!-- Definition du classpath du projet -->
  <path id="projet.classpath">
    <fileset dir="${projet.lib.dir}">
      <include name="*.jar"/>
    </fileset>
    <pathelement location="${projet.bin.dir}" />
  </path>

  <!-- Compilation des classes du projet -->
  <target name="compile" description="Compilation des classes">
    <javac srcdir="${projet.sources.dir}"
              destdir="${projet.bin.dir}"
                      debug="on"
```

```
                            optimize="off"
                            deprecation="on">
      <classpath refid="projet.classpath"/>
    </javac>
  </target>
</project>
```

Résultat :

```
C:\java\test\testhibernate>antBuildfile: build.xmlcompile:
    [javac] Compiling 1 source file to C:\java\test\testhibernate\bin
    [javac] C:\java\test\testhibernate\src\TestHibernate1.java:9: cannot resolve symbol
    [javac] symbol : class configuration
    [javac] location: class TestHibernate1
    [javac] Configuration config = new configuration();
    [javac] ^
    [javac] 1 error

BUILD FAILED
file:C:/java/test/testhibernate/build.xml:22: Compile failed; see the compiler e
rror output for details.

Total time: 9 seconds
```

69.4.8. javadoc

La tâche <javadoc> permet de demander la génération de la documentation au format javadoc des classes incluses dans une arborescence de répertoires.

Cette tâche possède plusieurs attributs dont les principaux sont :

Attribut	Rôle
sourcepath	précise le répertoire de base qui contient les sources dont la documentation est à générer
destdir	précise le répertoire qui va contenir les fichiers de documentation générés

Exemple :

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<project name="Test avec Ant" default="javadoc" basedir=".">
  <!-- =================================================-->
  <!-- Génération de la documentation Javadoc          -->
  <!-- =================================================-->
  <target name="javadoc">
    <javadoc sourcepath="src"
             destdir="doc" >
      <fileset dir="src" defaultexcludes="yes">
        <include name="**" />
      </fileset>
    </javadoc>
  </target>
</project>
```

Résultat :

```
C:\java\test\testant>ant
Buildfile: build.xml

javadoc:
  [javadoc] Generating Javadoc
  [javadoc] Javadoc execution
  [javadoc] Loading source file C:\java\test\testant\src\MaClasse.java...
  [javadoc] Constructing Javadoc information...
  [javadoc] Standard Doclet version 1.4.2_02
  [javadoc] Building tree for all the packages and classes...
```

```
  [javadoc] Building index for all the packages and classes...
  [javadoc] Building index for all classes...

BUILD SUCCESSFUL
Total time: 9 seconds
```

69.4.9. jar

La tâche <jar> permet la création d'une archive de type jar.

Cette tâche possède plusieurs attributs dont les principaux sont :

Attribut	Rôle
jarfile	nom du fichier .jar à créer
basedir	précise de répertoire qui contient les éléments à ajouter dans l'archive
compress	précise si le contenu de l'archive doit être compressé ou non. La valeur par défaut est true
manifest	précise le fichier manifest qui sera utilisé dans l'archive

Exemple :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
  <project name="Test avec Ant" default="packaging" basedir=".">

  <!-- =================================================================== -->
  <!-- Génération de l'archive jar                                       -->
  <!-- =================================================================== -->
  <target name="packaging">
    <jar jarfile="test.jar" basedir="src" />
  </target>
</project>
```

Résultat :

```
C:\java\test\testant>ant
Buildfile: build.xml

packaging:
      [jar] Building jar: C:\java\test\testant\test.jar

BUILD SUCCESSFUL
Total time: 2 seconds
```

La suite de ce chapitre sera développée dans une version future de ce document

70. Maven

Maven permet de faciliter et d'automatiser certaines tâches de la gestion d'un projet Java.

Le site officiel est http://maven.apache.org

Il permet notamment :

- d'automatiser certaines tâches : compilation, tests unitaires et déploiement des applications qui composent le projet
- de gérer des dépendances vis à vis des bibliothèques nécessaires au projet
- de générer des documentations concernant le projet

Au premier abord, il est facile de croire que Maven fait double emploi avec Ant. Ant et Maven sont tous les deux développés par le groupe Jakarta, ce qui prouve bien que leur utilité n'est pas aussi identique.

Ant dont le but est d'automatiser certaines tâches répétitives est plus ancien que Maven. Maven propose non seulement ces fonctionnalités mais en propose de nombreuses autres.

Pour gérer les dépendances du projet vis à vis de bibliothèques, Maven utilise un ou plusieurs repositorys qui peuvent être locaux (.maven/repository) ou distants (http://www.ibiblio.org/maven par défaut)

Maven est extensible grâce à un mécanisme de plug in qui permet d'ajouter des fonctionnalités.

70.1. L'installation

Il faut télécharger le fichier maven-1.0-rc2.exe sur le site de Maven et l'exécuter.

Un assistant permet de fournir les informations concernant l'installation :

- sur la page « Licence Agreement » : lire la licence et si vous l'acceptez cliquer sur le bouton « I Agree ».
- sur la page « Installations Options » : sélectionner les éléments à installer et cliquer sur le bouton « Next ».
- sur la page « Installation Folder » : sélectionner le répertoire dans lequel Maven va être installé et cliquer sur le bouton « Install ».
- une fois les fichiers copiés, il suffit de cliquer sur le bouton « Close ».

Sous Windows, un élément de menu nommé « Apache Software Foundation / Maven 1.0-rc2 » est ajouté dans le menu « Démarrer / Programmes ».

Pour utiliser Maven, la variable d'environnement système nommée MAVEN_HOME doit être définie avec comme valeur le chemin absolu du répertoire dans lequel Maven est installé. Par défaut, cette variable est configurée automatiquement lors de l'installation sous Windows.

Il est aussi particulièrement pratique d'ajouter le répertoire %MAVEN_HOME%/bin à la variable d'environnement PATH. Maven étant un outil en ligne de commande, cela évite d'avoir à saisir son chemin complet lors de son exécution.

Enfin, il faut créer un repository local en utilisant la commande ci dessous dans une boîte de commandes DOS :

Exemple :

```
C:\>install_repo.bat %HOMEDRIVE%%HOMEPATH%
```

Pour s'assurer de l'installation correcte de Maven, il suffit de saisir la commande :

Exemple :

```
C:\>maven -v
 __  __
|  \/  |__ _Apache__ ___
| |\/| / _` \ V / -_) ' \  ~ intelligent projects ~
|_|  |_\__,_|\_/\___|_||_|  v. 1.0-rc2
C:\>
```

Lors de la première exécution de Maven, ce dernier va constituer le repository local (une connexion internet est nécessaire).

Exemple :

```
|  \/  |__ _Apache__ ___
| |\/| / _` \ V / -_) ' \  ~ intelligent projects ~
|_|  |_\__,_|\_/\___|_||_|  v. 1.0-rc2
Le rÚpertoire C:\Documents and Settings\Administrateur\.maven\repository n'exist
e pas. Tentative de crÚation.
Tentative de tÚlÚchargement de commons-lang-1.0.1.jar.
................................
.
Tentative de tÚlÚchargement de commons-net-1.1.0.jar.
..............................................................................
........
.
Tentative de tÚlÚchargement de dom4j-1.4-dev-8.jar.
..............................................................................
..............................................................................
..............................................................................
............................................................
.
Tentative de tÚlÚchargement de xml-apis-1.0.b2.jar.
.......................................................
```

70.2. Les plug-ins

Toutes les fonctionnalités de Maven sont proposées sous la forme de plug-ins.

Le fichier maven.xml permet de configurer les plug-ins installés.

70.3. Le fichier project.xml

Maven est orienté projet, donc le projet est l'entité principale gérée par Maven. Il est nécessaire de fournir à Maven une description du projet (Project descriptor) sous la forme d'un document XML nommé project.xml et situé à la racine du répertoire contenant le projet.

Exemple : un fichier minimaliste

```
  <project>
```

```
    <id>P001</id>
    <name>TestMaven</name>
    <currentVersion>1.0</currentVersion>
    <shortDescription>Test avec Maven</shortDescription>

    <developers>
      <developer>
        <name>Jean Michel D.</name>
        <id>jmd</id>
        <email>jmd@test.fr</email>
      </developer>
    </developers>

    <organization>
      <name>Jean-Michel</name>
    </organization>

  </project>
```

Il est possible d'inclure la valeur d'un tag défini dans le document dans un autre tag.

Exemple :

```
  ...
  <shortDescription>${pom.name} est un test avec Maven</shortDescription>
  ...
```

Il est possible d'hériter d'un fichier project.xml existant dans lequel des caractéristiques communes à plusieurs projets sont définies. La déclaration dans le fichier du fichier père se fait avec le tag <extend>. Dans le fichier fils, il suffit de redéfinir ou de définir les tags nécessaires.

70.4. L'exécution de Maven

Maven s'utilise en ligne de commande sous la forme suivante :

Maven plugin:goal

Il faut exécuter Maven dans le répertoire qui contient le fichier project.xml.

Si les paramètres fournis ne sont pas corrects, une exception est levée :

Exemple :

```
C:\java\test\testmaven>maven compile
 __  __
|  \/  |__ _Apache__ ___
| |\/| / _` \ V / -_) ' \  ~ intelligent projects ~
|_|  |_\__,_|\_/\___|_||_|  v. 1.0-rc2
com.werken.werkz.NoSuchGoalException: No goal [compile]
        at com.werken.werkz.WerkzProject.attainGoal(WerkzProject.java:190)
        at org.apache.maven.plugin.PluginManager.attainGoals(PluginManager.java:
531)
        at org.apache.maven.MavenSession.attainGoals(MavenSession.java:265)
        at org.apache.maven.cli.App.doMain(App.java:466)
        at org.apache.maven.cli.App.main(App.java:1117)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.
java:39)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAcces
sorImpl.java:25)
        at java.lang.reflect.Method.invoke(Method.java:324)
        at com.werken.forehead.Forehead.run(Forehead.java:551)
        at com.werken.forehead.Forehead.main(Forehead.java:581)
```

```
Total time: 4 seconds
Finished at: Tue May 18 14:17:18 CEST 2004
```

Pour obtenir une liste complète des plug-ins à disposition de Maven, il suffit d'utiliser la commande maven -g

Voici quelques uns des nombreux plug-ins avec leurs goals principaux :

Plug in	Goal	Description
ear	ear	construire une archive de type ear
	deploy	déployer un fichier ear dans un serveur d'application
ejb	ejb	
	deploy	
jalopy	format	
java	compile	compiler des sources
	jar	créer une archive de type .jar
javadoc		
jnlp		
pdf		générer la documentation du projet au format PDF
site	generate	générer le site web du projet
	deploy	copier le site web sur un serveur web
test	match	exécuter des tests unitaires
war	init	
	war	
	deploy	

La commande maven clean permet d'effacer tous les fichiers générés par Maven.

Exemple :

```
C:\java\test\testmaven>maven clean
 __  __
|  \/  |__ _Apache__ ___
| |\/| / _` \ V / -_) ' \  ~ intelligent projects ~
|_|  |_\__,_|\_/\___|_||_|  v. 1.0-rc2
build:start:
clean:clean:
    [delete] Deleting directory C:\java\test\testmaven\target
    [delete] Deleting: C:\java\test\testmaven\velocity.log
BUILD SUCCESSFUL
Total time: 2 minutes 7 seconds
Finished at: Tue May 25 14:19:03 CEST 2004
C:\java\test\testmaven>
```

70.5. La génération du site du projet

Maven propose une fonctionnalité qui permet de générer automatique un site web pour le projet regroupant un certain nombre d'informations utiles le concernant.

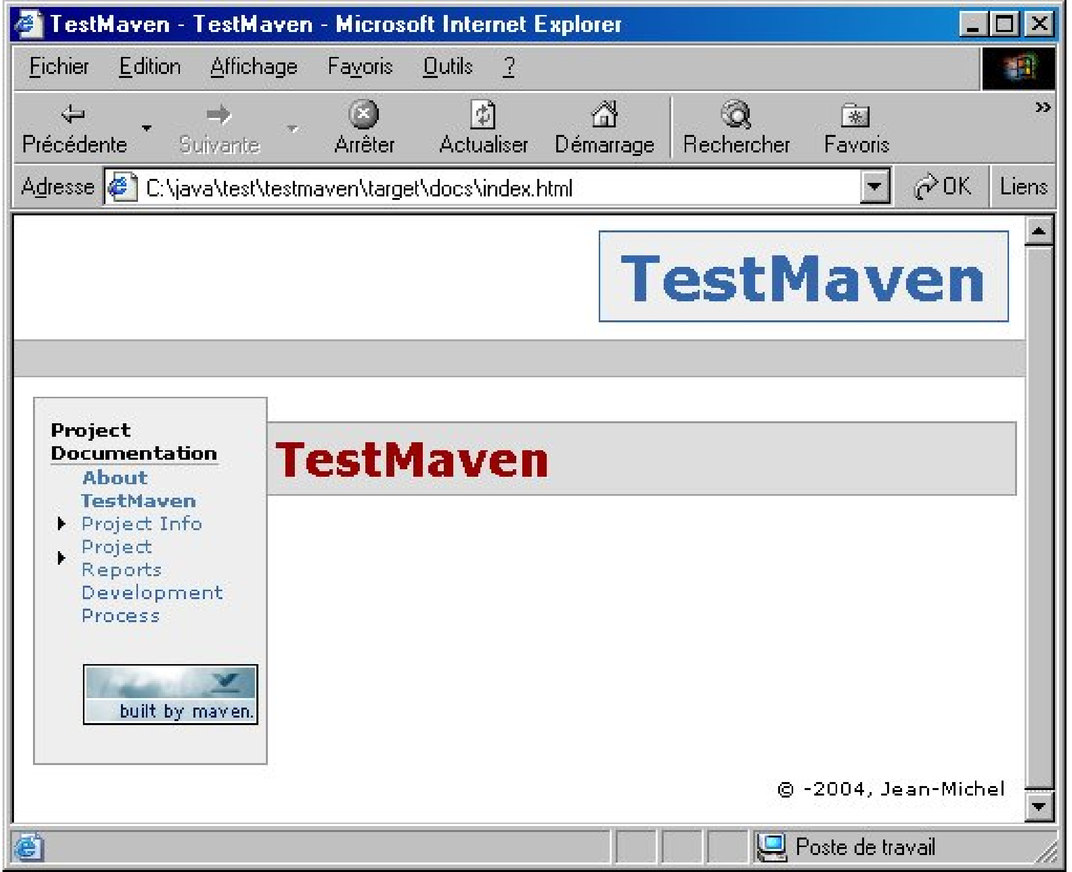
Pour demander la génération du site, il suffit de saisir la commande

maven site:generate

Lors de l'exécution de cette commande, un répertoire target/docs est créé contenant les différents éléments du site.

Exemple :

```
C:\java\test\testmaven\target\docs>dir
 Le volume dans le lecteur C s'appelle MACHINE
 Le numéro de série du volume est 3T78-19E4
 Répertoire de C:\java\test\testmaven\target\docs
25/05/2004  14:21       <DIR>          .
25/05/2004  14:21       <DIR>          ..
25/05/2004  14:21       <DIR>          apidocs
25/05/2004  14:21                5 961 checkstyle-report.html
25/05/2004  14:21                1 637 cvs-usage.html
25/05/2004  14:21                1 954 dependencies.html
25/05/2004  14:21       <DIR>          images
25/05/2004  14:21                1 625 index.html
25/05/2004  14:21                1 646 issue-tracking.html
25/05/2004  14:21                3 119 javadoc.html
25/05/2004  14:21                9 128 jdepend-report.html
25/05/2004  14:21                2 494 license.html
25/05/2004  14:21                5 259 linkcheck.html
25/05/2004  14:21                1 931 mail-lists.html
25/05/2004  14:21                4 092 maven-reports.html
25/05/2004  14:21                3 015 project-info.html
25/05/2004  14:21       <DIR>          style
25/05/2004  14:21                2 785 task-list.html
25/05/2004  14:21                3 932 team-list.html
25/05/2004  14:21       <DIR>          xref
              14 fichier(s)          48 578 octets
               6 Rép(s)     207 151 616 octets libres
```



Par défaut, le site généré contient un certain nombre de pages accessibles via le menu de gauche.

La partie « Project Info » regroupe trois pages : la mailing liste, la liste des développeurs et les dépendances du projet.

La partie « Project report » permet d'avoir accès à des comptes rendus d'exécution de certaines tâches : javadoc, tests unitaires, ... Certaines de ces pages ne sont générées qu'en fonction des différents éléments générés par Maven.

Le contenu du site pourra donc être réactualisé facilement en fonction des différents traitements réalisés par Maven sur le projet.

70.6. La compilation du projet

Dans le fichier project.xml, il faut rajouter un tag <build> qui va contenir les informations pour la compilation des éléments du projet.

Les sources doivent être contenues dans un répertoire dédié, par exemple src

Exemple :

```
  ...
    <build>
        <sourceDirectory>
          ${basedir}/src
        </sourceDirectory>
      </build>
  ...
```

Pour demander la compilation à Maven, il faut utiliser la commande

Exemple :

```
Maven java :compile
C:\java\test\testmaven>maven java:compile
 __  __
|  \/  |__ _Apache__ ___
| |\/| / _` \ V / -_) ' \  ~ intelligent projects ~
|_|  |_\__,_|\_/\___|_||_|  v. 1.0-rc2
Tentative de tÚlÚchargement de commons-jelly-tags-antlr-20030211.143720.jar.
.....
.
build:start:
java:prepare-filesystem:
    [mkdir] Created dir: C:\java\test\testmaven\target\classes
java:compile:
    [echo] Compiling to C:\java\test\testmaven/target/classes
    [javac] Compiling 1 source file to C:\java\test\testmaven\target\classes
BUILD SUCCESSFUL
Total time: 12 seconds
Finished at: Tue May 18 14:19:12 CEST 2004
```

Le répertoire « target/classes » est créé à la racine du répertoire du projet. Les fichiers .class issus de la compilation sont stockés dans ce répertoire.

La commande maven jar permet de demander la génération du packaging de l'application.

Exemple :

```
build:start:
java:prepare-filesystem:
java:compile:
    [echo] Compiling to C:\java\test\testmaven/target/classes
java:jar-resources:
test:prepare-filesystem:
    [mkdir] Created dir: C:\java\test\testmaven\target\test-classes
```

```
    [mkdir] Created dir: C:\java\test\testmaven\target\test-reports
test:test-resources:
test:compile:
    [echo] No test source files to compile.
test:test:
    [echo] No tests to run.
jar:jar:
    [jar] Building jar: C:\java\test\testmaven\target\P001-1.0.jar
BUILD SUCCESSFUL
Total time: 2 minutes 42 seconds
Finished at: Tue May 18 14:25:39 CEST 2004
```

Par défaut, l'appel à cette commande effectue une compilation des sources, un passage des tests unitaires si il y en a et un appel à l'outil jar pour réaliser le packaging.

Le nom du fichier jar créé est composé de l'id du projet et du numéro de version. Il est stocké dans le répertoire racine du projet.

71. Tomcat

Chapitre 71

71.1. La présentation de Tomcat

Tomcat est un conteneur d'applications web diffusé en open source sous une licence Apache. C'est aussi l'implémentation de référence des spécifications servlets et JSP implémentées dans les différentes versions de Tomcat.

En tant qu'implémentation de référence, facile à mettre en oeuvre et riche en fonctionnalités, Tomcat est quasi incontournable dans les environnements de développements. Les qualités de ces dernières versions lui permettent d'être de plus en plus utilisé dans des environnements de production.

71.1.1. L'historique des versions

Il existe plusieurs versions de Tomcat qui mettent en oeuvre des versions différentes des spécifications des servlets et des JSP :

Version de Tomcat	Version Servlet	Version JSP
3.0, 3.1, 3.2, 3.3	2.2	1.1
4.0, 4.1	2.3	1.2
5.0	2.4	2.0
6.0	2.5	2.1

Tomcat 3.x (version initiale)

- implémente les spécifications Servlet 2.2 et JSP 1.1
- rechargement des servlets
- fonctionnalités HTTP de base.

Tomcat 4.x

- implémente les spécifications Servlet 2.3 et JSP 1.2
- nouveau conteneur de servlets Catalina
- nouveau moteur JSP Jasper
- le connecteur Coyote
- utilisation de JMX,
- application d'administration développée en Struts

Tomcat 5.x

- implémente les spécifications Servlet 2.4 et JSP 2.0
- performances améliorées
- wrapper natifs pour Windows et Unix
- amélioration du traitement des JSP

Tomcat 5.5 nécessite un J2SE 5.0 pour fonctionner. Un module dédié permet d'utiliser Tomcat 5.5 avec un JDK 1.4 mais cela n'est pas recommandé.

Tomcat 5.5 utilise le compilateur d'Eclipser pour compiler les JSP : il n'est donc plus nécessaire d'installer un JDK pour faire fonctionner Tomcat, un JRE suffit.

La configuration de Tomcat 5.5 est différente de celle de Tomcat 5.0 sur de nombreux points

Tomcat 6.x

- implémente la version 2.5 des spécifications des servlets (JSR-154)
- implémente la version 2.1 des spécifications des JSP (JSR-245)
- implémente Unified EL (Unified Expression Langage)
- utilise Java 5
- amélioration de l'usage mémoire

71.2. L'installation

Tomcat est une application écrite en Java, il est possible de l'installer et de l'exécuter sous tous les environnements disposant d'une machine virtuelle Java : un JRE et même un JDK pour certaine ancienne version est un pré requis pour permettre sont exécution.

Pour les versions de Tomcat nécessitant un JDK, il faut que la variable d'environnement JAVA_HOME soit définie avec comme valeur le répertoire d'installation du JDK. Ceci permet notamment à Tomcat de trouver le compilateur Java pour compiler les JSP.

L'installation de Tomcat de façon universelle se fait simplement :

- Tomcat est fourni dans une archive de type zip qu'il faut télécharger
- décompresser l'archive dans un répertoire du système
- Il est préférable de définir la variable d'environnement système CATALINA_HOME qui possède comme valeur le répertoire d'installation de Tomcat

Sous Windows, Tomcat propose un package d'installation qui va permettre en plus :

- De demander et configurer le port du connecteur http à utiliser
- De créer une entrée dans le menu « démarrer / Programme » avec des raccourcis vers quelques fonctionnalités
- D'exécuter Tomcat uniquement sous la forme d'un service Windows

71.2.1. L'installation de Tomcat 3.1 sous Windows 98

Il est possible de récupérer tomcat sur le site de Jakarta. Il faut choisir la version stable de préférence et le répertoire bin pour récupérer le fichier jakarta-tomcat.zip.

Il faut ensuite décompresser le fichier dans un répertoire du système par exemple dans C:\. L'archive est décompressée dans un répertoire nommé jakarta-tomcat

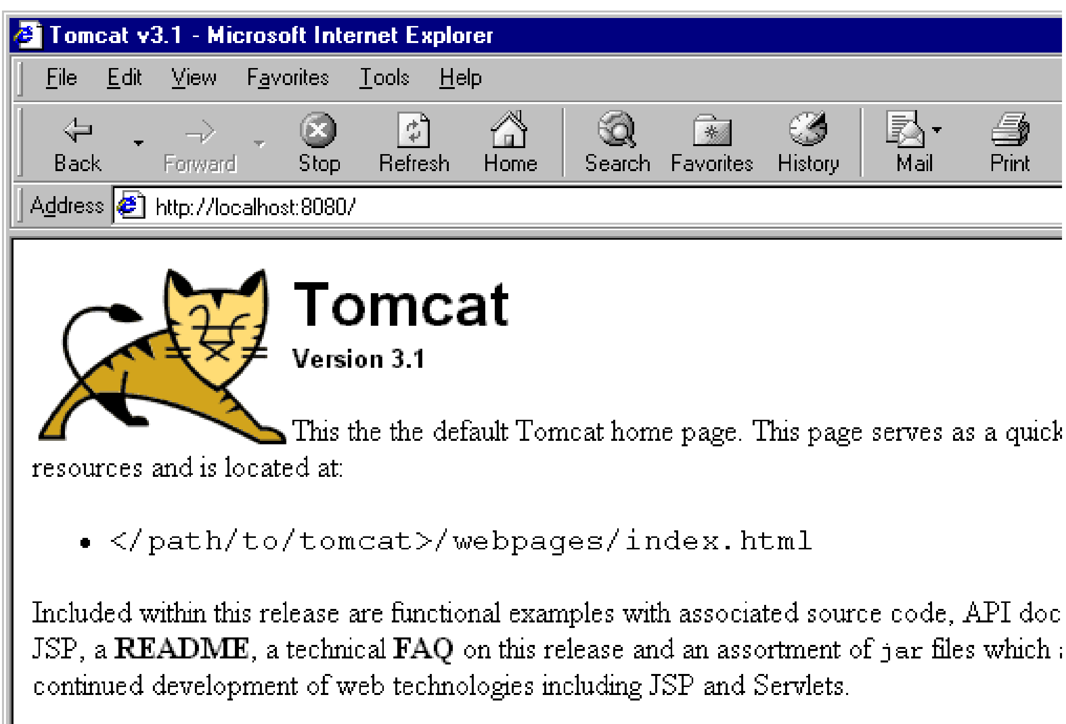
Dans une boite DOS, assigner le répertoire contenant Tomcat dans une variable d'environnement TOMCAT_HOME. Le plus simple est de l'ajouter dans le fichier autoexec.bat.

Exemple :

```
set TOMCAT_HOME=c:\jakarta-tomcat
```

Pour lancer Tomcat, il faut exécuter le fichier startup.bat dans le répertoire TOMCAT_HOME\bin

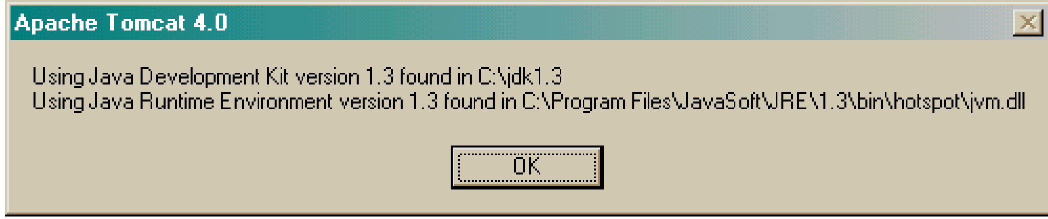
Pour vérifier que Tomcat s'exécute correctement, il faut saisir l'url http://localhost:8080 dans un browser. La page d'accueil de Tomcat s'affiche.



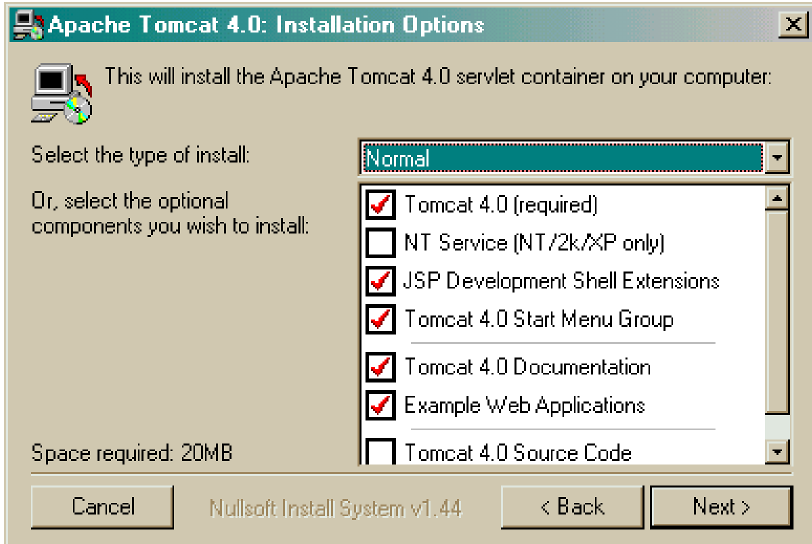
Le script %TOMCAT_HOME%\bin\shutdown.bat permet de stopper Tomcat.

71.2.2. L'installation de Tomcat 4.0 sur Windows 98

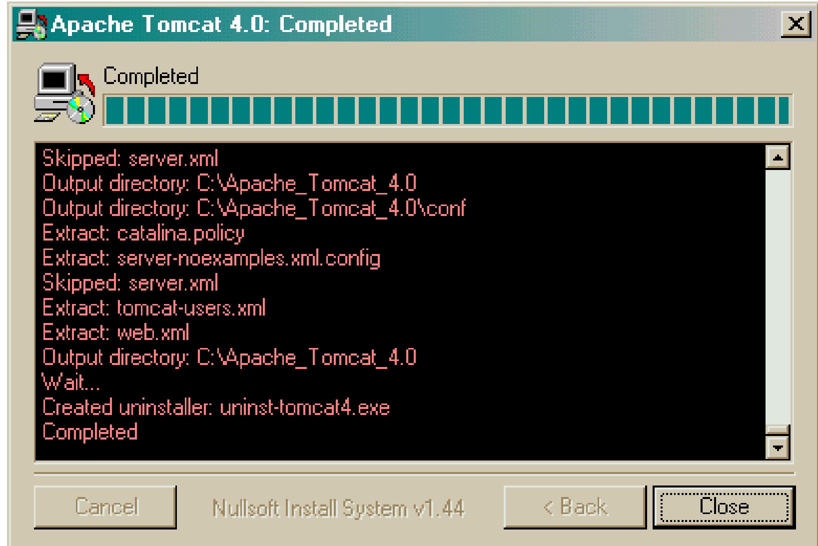
Il suffit de télécharger et d'exécuter le programme jakarta-tomcat-4.0.exe



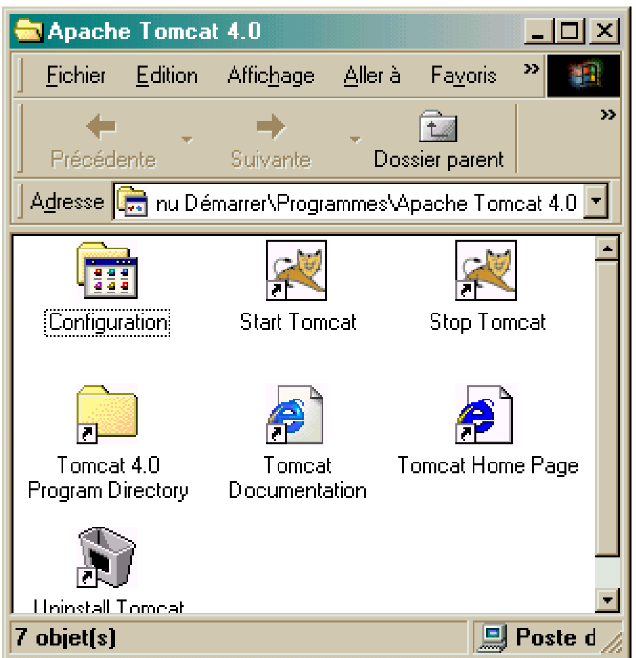
L'assistant affiche la licence, puis permet de sélectionner les options d'installation et le répertoire d'installation.



L'assistant copie les fichiers.



Un ensemble de raccourcis est créé dans l'option "Apache Tomcat 4.0" du menu "Démarrer/Programmes"



Il faut définir la variable d'environnement système JAVA_HOME qui doit avoir comme valeur le chemin absolu du répertoire d'installation du J2SDK.

Pour la version 4.1, il faut télécharger le fichier jakarta-tomcat-4.1.29.exe sur le site http://jakarta.apache.org/site/binindex.cgi

71.2.3. L'installation de Tomcat 5.0 sur Windows

Il faut télécharger le fichier jakarta-tomcat-5.0.16.exe sur le site http://jakarta.apache.org/site/binindex.cgi

La version 5 utilise un programme d'installation standard guidé par un assistant qui propose les étapes suivantes :

- la page d'accueil s'affiche, cliquez sur le bouton « Next »
- la page d'acceptation de la licence (« Licence agreement ») s'affiche, lire la licence et si vous l'acceptez, cliquez sur le bouton « I Agree »
- la page de sélection des composants à installer (« Choose components ») s'affiche, il faut sélectionner ou non chacun des composants ou utiliser un type d'installation qui contient une pré configuration, cliquez sur le bouton « Next »
- la page de sélection du répertoire d'installation (« Choose install location ») s'affiche, sélectionner le répertoire de destination et cliquez sur le bouton « Next »
- la page de configuration (« Basic configuration ») s'affiche et permet de définir le port du connecteur http (8080 par défaut) utilisé, le nom et le mot de passe de l'administrateur. Saisissez ces informations et cliquez sur le bouton « Next »
- la page de sélection du chemin de la JVM (« Java Virtual Machine ») permet de sélectionner le chemin du JRE. Cliquez sur le bouton « Install »
- l'installation s'effectue

la page de fin d'affiche. Une case à cocher permet de demander le lancement de Tomcat. Cliquez sur le bouton « Finish »

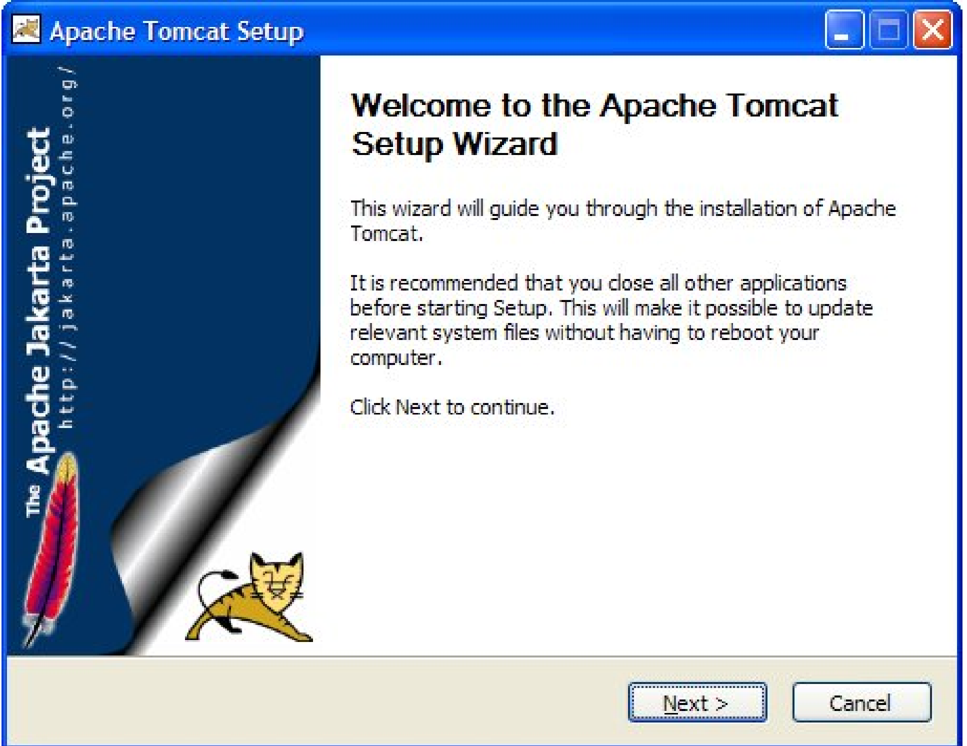
71.2.4. L'installation de Tomcat 5.5 sous Windows avec l'installer

L'url pour télécharger la version 5.5 de Tomcat est http://tomcat.apache.org/download-55.cgi
Attention : Tomcat 5.5 est packagé différemment par rapport à ces précédentes versions : les différents modules qui composent Tomcat sont fournis séparément. Ceci permet d'installer uniquement les modules souhaités de Tomcat notamment dans un environnement de production.
Les modules sont :

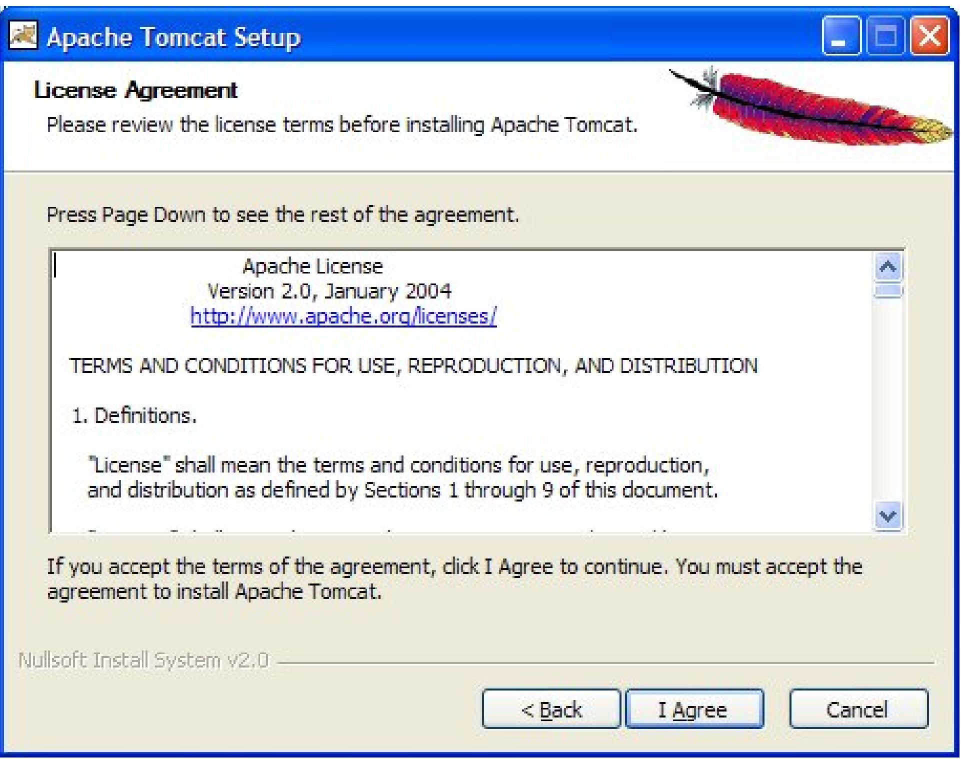
- Core : ce module contient le serveur Tomcat
- Deployer : ce module contient le TCD (Tomcat Client Deployer) qui utilise Ant pour compiler, valider et déployer une application web
- Embedded : ce module contient une version embarquée de Tomcat (pour l'intégrer dans une autre application)
- Administration web application : ce module contient l'application web d'administration de Tomcat
- JDK 1.4 Compatibility Package : ce mode doit être utilisé pour exécuter Tomcat 5.5 avec un JDK 1.4
- Documentation : ce module contient la documentation seule (ce module est inclus dans le module Core)

Téléchargez le setup de la dernière version de Tomcat 5.5 (par exemple apache-tomcat-5.5.23.exe) et exécuter ce fichier.

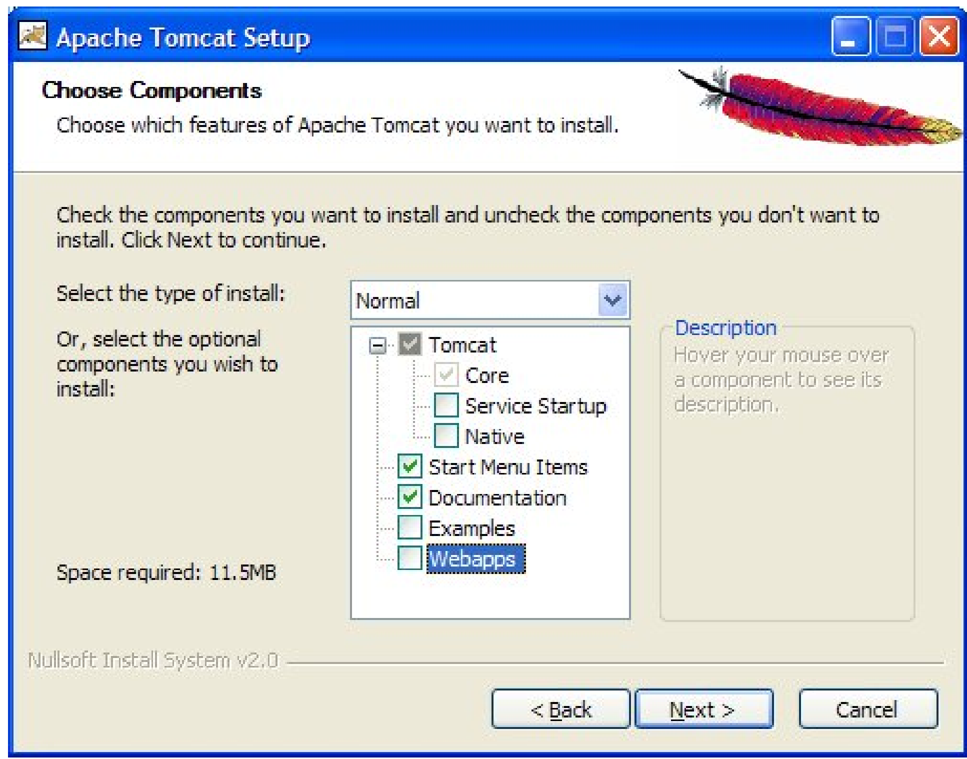
Attention : l'installer ne permet que l'exécution de Tomcat sous la forme d'un service Windows.



Cliquez sur le bouton « Next »



Lisez la licence et si vous l'acceptez cliquez sur le bouton « I Agree »



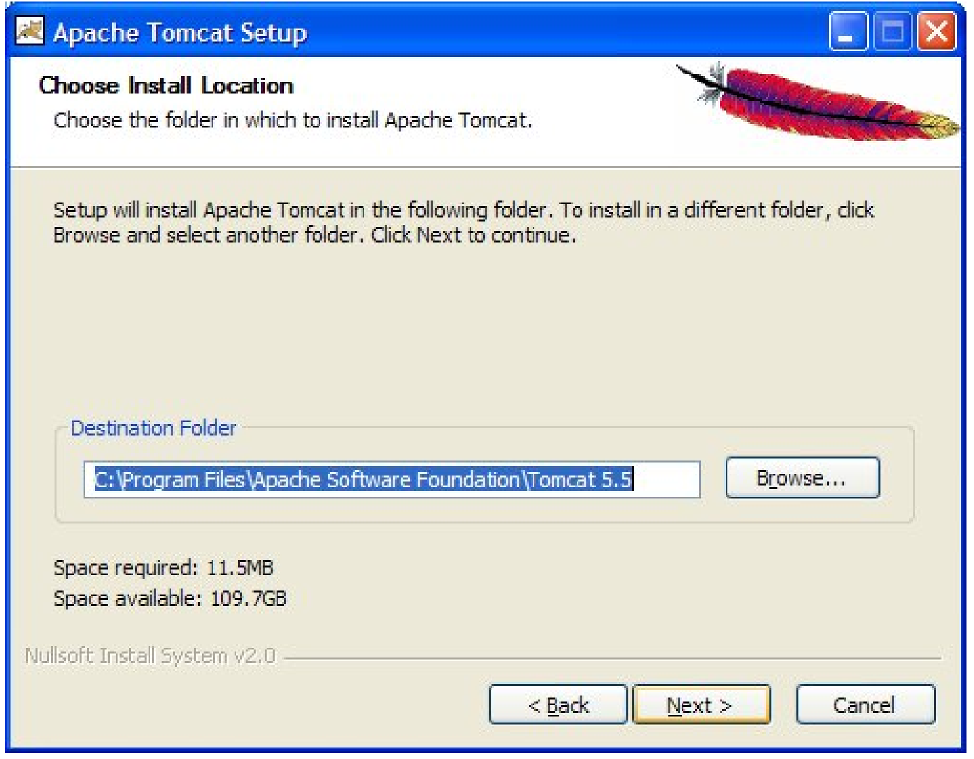
Cette page permet de sélectionner les composants à installer en sélectionnant le type d'installation. Le type custom permet une sélection de chaque composant.

Le composant « Service Startup » permet de demander le démarrage automatique du service Tomcat

Le composant « Native » permet d'installer certaines librairies natives

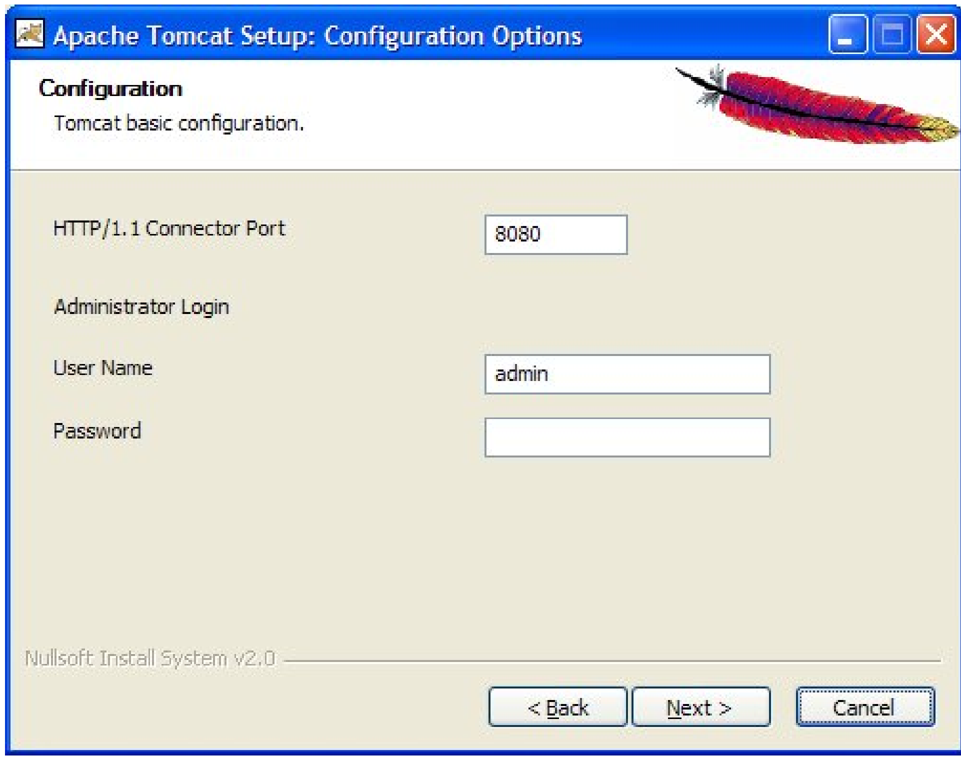
Le composant « Start Menu Items » permet de créer une entrée dans le menu « Démarrer / Programme » avec des raccourcis vers certaines fonctionnalités.

Sélectionnez le type d'installation et les composants à installer si besoin et cliquez sur le bouton « Next »

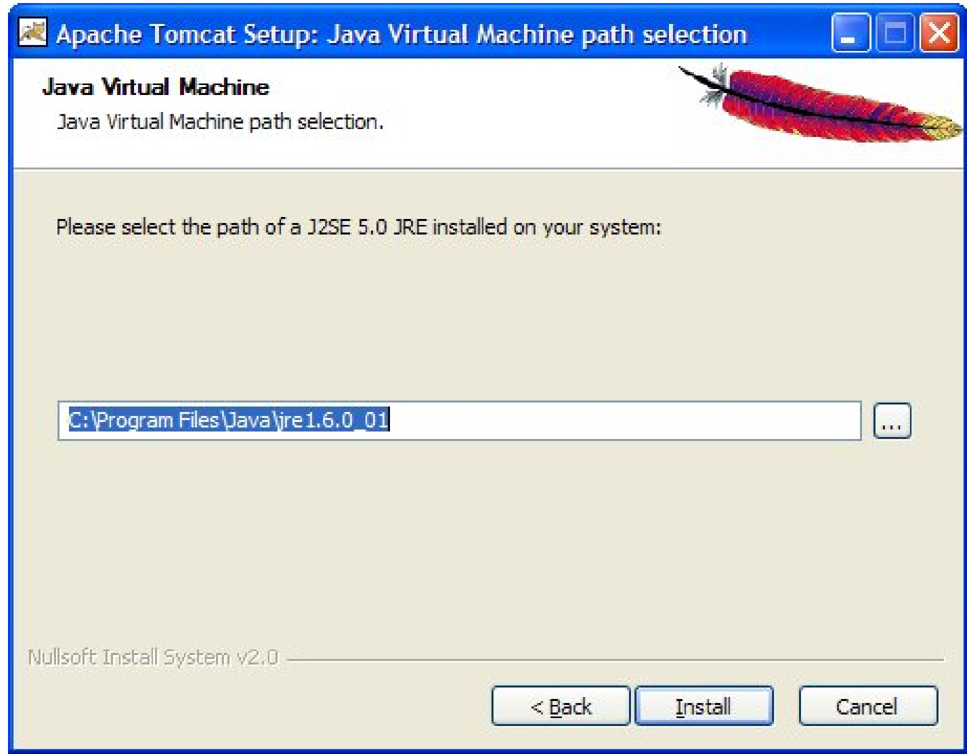


Cette page permet de sélectionner le répertoire d'installation.

Sélectionnez un autre répertoire si besoin et cliquez sur le bouton « Next »

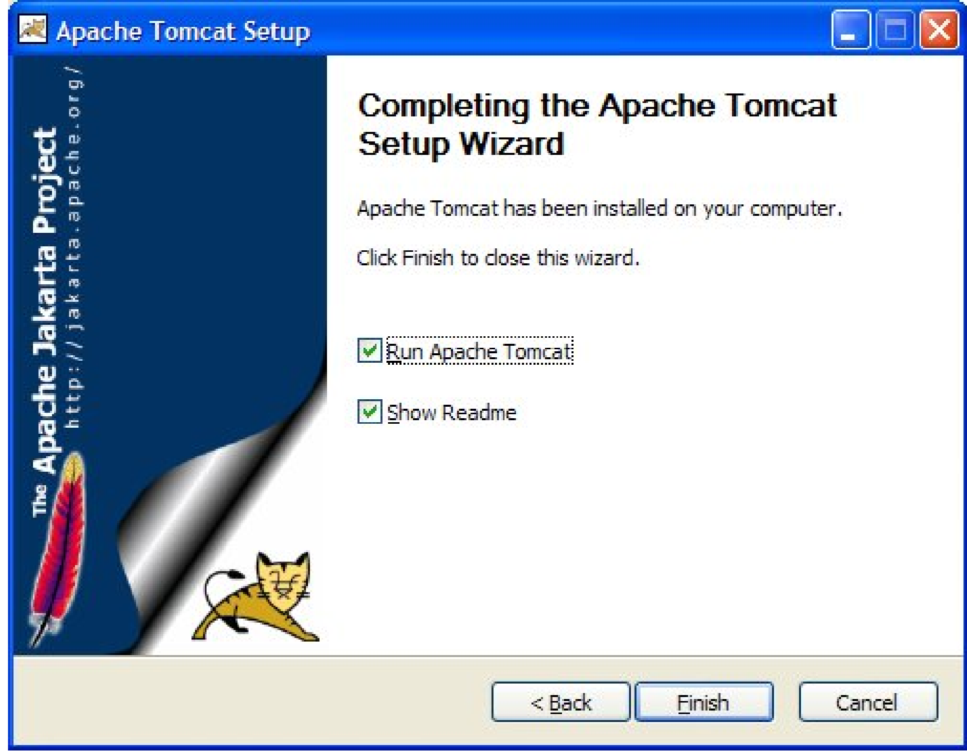


Cette page permet de préciser le port du connecteur http à utiliser (8080 par défaut) et de préciser les informations de login de l'administrateur de Tomcat



La page suivante permet de préciser le chemin du JRE 5.0 minimum à utiliser.

Cliquez sur le bouton « Install » pour démarrer l'installation de Tomcat.



Cliquez sur le bouton « Finish »

L'installation des autres modules de Tomcat se fait en les décompressant dans le répertoire d'installation de Tomcat en s'assurant que celui-ci est arrêté.

71.2.5. L'installation Tomcat 6.0 sous Windows avec l'installer

La procédure est similaire à celle de la version 5.5 de Tomcat.
L'url pour télécharger la version 6.0 de Tomcat est http://tomcat.apache.org/download-60.cgi
Exécutez le fichier téléchargé, par exemple apache-tomcat-6.0.10.exe

L'installation se fait via un assistant :

- Sur la page « Welcome to Apache Tomcat Setup Wizard », cliquez sur le bouton « Next »
- Sur la page « Licence Agreement », lisez la licence et si vous l'acceptez cliquez sur le bouton « I Agree »
- Sur la page « Choose Components », sélectionnez les éléments à installer et cliquez sur le bouton « Next »
- Sur la page « Choose Install Location », cliquez sur le bouton « Next »
- Sur la page « Configuration », modifier au besoin le numéro de port du connecteur http (qui sera donc utilisé dans les url) et les informations de login de l'administrateur et cliquez sur le bouton « Next »
- Sur la page « Java Virtual Machine », cliquez sur le bouton « Install »
- Sur la page « Completing the Apache Tomcat Setup Wizard », cliquez sur le bouton Finish

71.2.6. La structure des répertoires

Le répertoire d'installation de Tomcat contient plusieurs répertoires.

71.2.6.1. 1.2.6.1 La structure des répertoires de Tomcat 4

Le répertoire où est installé Tomcat est composé de l'arborescence suivante :

- bin : contient un ensemble de scripts pour la mise en oeuvre de Tomcat

* common : le sous répertoire lib contient les bibliothèques utilisées par Tomcat et mises à disposition de toutes les applications qui seront exécutées dans Tomcat
* conf : contient des fichiers de propriétés notamment les fichiers server.xml, tomcat-users.xml et le fichier par défaut web.xml
* logs : contient les journaux d'exécution
* server : contient des bibliothèques utilisées par Tomcat et l'application web d'administration de Tomcat
* temp : est un répertoire temporaire utilisé lors des traitements
* webapps : contient les applications web exécutées sous Tomcat
* work : contient le résultat de la compilation des JSP en servlets

71.2.6.2. La structure des répertoires de Tomcat 5

Le répertoire d'installation de Tomcat 5.x contient plusieurs répertoires :

* bin : scripts et exécutables pour gérer Tomcat
* common : bibliothèques et classes communes pour Catalina et les applications web
* conf : fichiers de configurations
* logs : journaux de Catalina et des applications web
* server : bibliothèques et classes utilisées uniquement par Catalina
* shared : bibliothèques et classes partagées par les applications web
* temp : répertoire de stockage de fichiers temporaires
* webapps : répertoire de déploiement des applications web
* work : répertoire de travail (répertoires et fichiers notamment pour la compilation des JSP)

Le répertoire conf contient en standard plusieurs fichiers de configuration :

Fichier	Rôle
catalina.policy	
catalina.properties	Configuration du chargement des classes par Tomcat
context.xml	Configuration par défaut utilisé par tous les contextes
logging.properties	Configuration des logs de Tomcat
server.xml	Configuration du serveur Tomcat
tomcat-users.xml	Contient les données utiles pour l'authentification et pour les habilitations (user et rôle)
web.xml	Descripteur de déploiement par défaut utilisé pour toutes les applications web avant de traiter le fichier des applications

Le répertoire logs est le répertoire par défaut des logs. Sa taille ne fait que croître : il est donc nécessaire de la surveiller notamment dans un environnement de production.

Remarque : l'utilisation du répertoire shared pour mettre des bibliothèques ou des classes est déconseillée. C'est une particularité de Tomcat : il est préférable d'utiliser le répertoire WEB-INF/classes pour les classes et WEB-INF/lib pour les bibliothèques de la webapp car c'est le standard.

71.2.6.3. La structure des répertoires de Tomcat 6

La structure des répertoires est similaire à celle de Tomcat 5 hormis pour les répertoires shared et server qui sont remplacés par un unique répertoire lib. Ce répertoire lib ne contient pas de sous répertoire lib et classes : il contient directement les bibliothèques.

71.3. L'exécution de Tomcat

Le lancement de Tomcat s'effectue en utilisant un script fourni dans le sous répertoire d'installation de Tomcat. Sous Windows, il est possible de lancer Tomcat sous la forme d'un service.

71.3.1. L'exécution sous Windows de Tomcat 4.0

Sous Windows, pour lancer Tomcat manuellement, il faut exécuter la commande startup.bat dans le sous répertoire bin du répertoire où est installé Tomcat. La commande shutdown.bat permet inversement de stopper l'exécution de Tomcat.

Par défaut, le serveur web intégré dans Tomcat utilise le port 8080 pour recevoir les requêtes HTTP. Pour vérifier la bonne installation de l'outil, il suffit d'ouvrir un navigateur et de demander l'URL : http://localhost:8080/

71.3.2. L'exécution sous Windows de Tomcat 5.0

En utilisant les scripts startup et shutdown

Pour lancer Tomcat, il suffit d'exécuter le script startup.bat du sous répertoire bin.

Pour arrêter Tomcat, il suffit d'exécuter le script shutdown.bat du sous répertoire bin.

Pour une utilisation en ligne de commande (sans IDE pour piloter Tomcat), il est pratique de créer un lien vers ces deux scripts, par exemple sur le bureau. L'avantage de les mettre sur le bureau est qu'il possible de leur assigner des raccourcis clavier.

En utilisant l'application tomcat5.exe

En lançant le programme bin/Tomcat5.exe du répertoire d'installation de Tomcat, Tomcat est lancé sous la forme d'un service : les messages de la console sont affichés dans la boîte Dos associé au processus.

```
C:\Program Files\Apache Software Foundation\Tomcat 5.0\bin\tomcat5.exe
1.6.0\bin;C:\Program Files\Sandcastle\ProductionTools\
10 mai 2007 11:55:56 org.apache.coyote.http11.Http11BaseProtocol init
INFO: Initialisation de Coyote HTTP/1.1 sur http-8088
10 mai 2007 11:55:56 org.apache.catalina.startup.Catalina load
INFO: Initialization processed in 781 ms
10 mai 2007 11:55:56 org.apache.catalina.core.StandardService start
INFO: Dúmarrage du service Catalina
10 mai 2007 11:55:56 org.apache.catalina.core.StandardEngine start
INFO: Starting Servlet Engine: Apache Tomcat/5.5.23
10 mai 2007 11:55:56 org.apache.catalina.core.StandardHost start
INFO: XML validation disabled
10 mai 2007 11:55:57 org.apache.coyote.http11.Http11BaseProtocol start
INFO: Dúmarrage de Coyote HTTP/1.1 sur http-8088
10 mai 2007 11:55:57 org.apache.jk.common.ChannelSocket init
INFO: JK: ajp13 listening on /0.0.0.0:8009
10 mai 2007 11:55:57 org.apache.jk.server.JkMain start
INFO: Jk running ID=0 time=0/31  config=null
10 mai 2007 11:55:57 org.apache.catalina.storeconfig.StoreLoader load
INFO: Find registry server-registry.xml at classpath resource
10 mai 2007 11:55:57 org.apache.catalina.startup.Catalina start
INFO: Server startup in 1157 ms
```

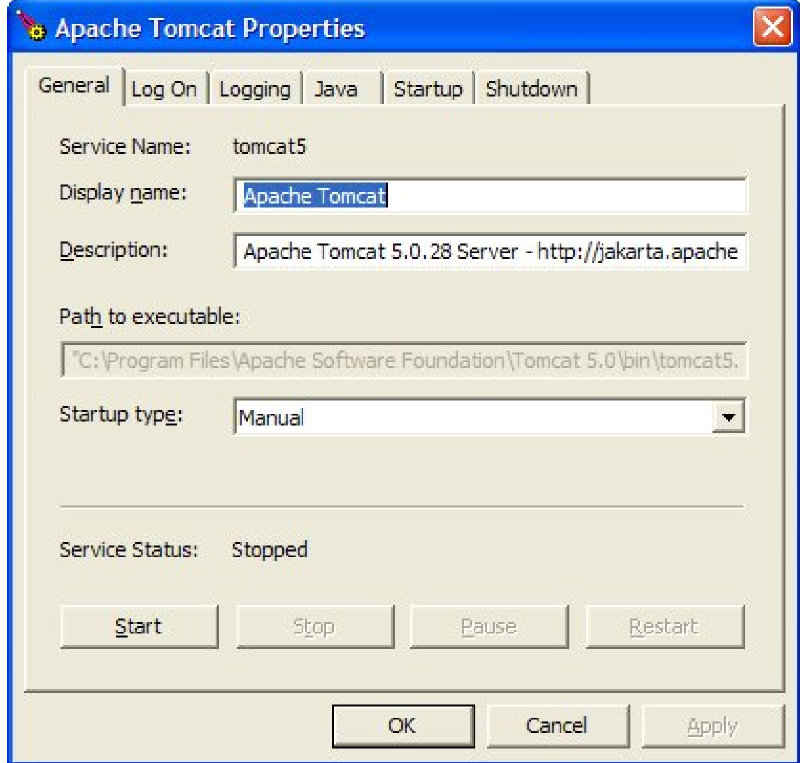
Tomcat apparaît dans les processus du gestionnaire de tâches.

Pour arrêter Tomcat dans ce cas, il faut fermer la fenêtre Dos : le serveur sera arrêté proprement.

En utilisant Tomcat en tant que service Windows

Le programme d'installation de Tomcat fourni un utilitaire supplémentaire qui permet d'exécuter et de gérer Tomcat en tant que service Windows.

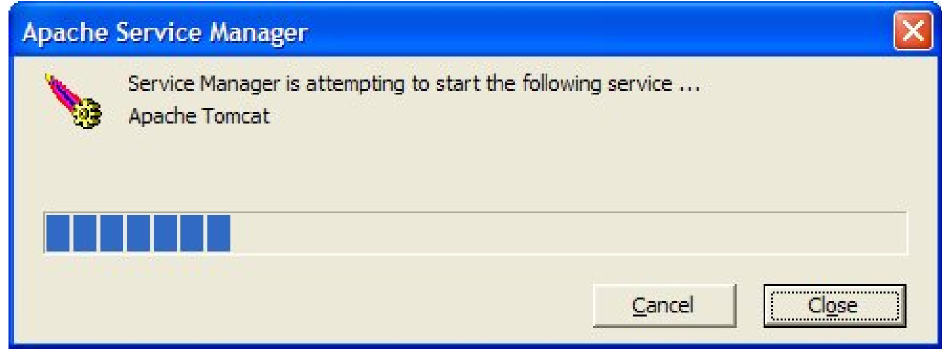
En lançant le programme bin/tomcat5w.exe du répertoire d'installation de Tomcat, une application qui permet de configurer et de gérer l'exécution de Tomcat sous la forme d'un service Windows est lancée.



Cette application permet de gérer Tomcat en tant que service Windows.

L'onglet « General » permet de gérer l'exécution de Tomcat sous la forme d'un service.

Pour gérer le statut du service de Tomcat, il suffit d'utiliser le bouton correspondant.



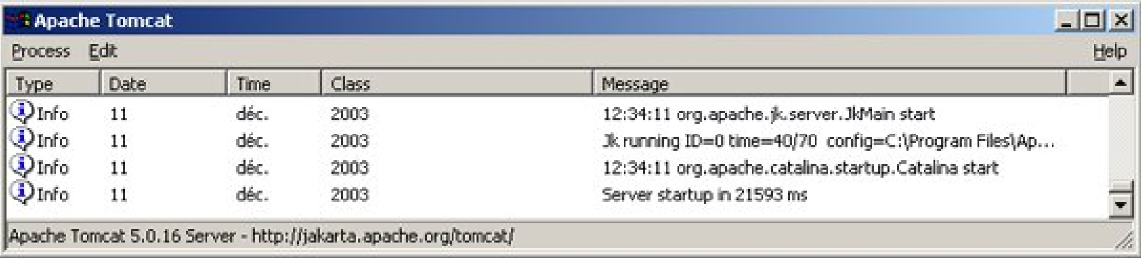
Les autres onglets permettent de préciser des paramètres d'exécution de Tomcat.

Après le démarrage de Tomcat, une icône apparaît dans la barre d'icône

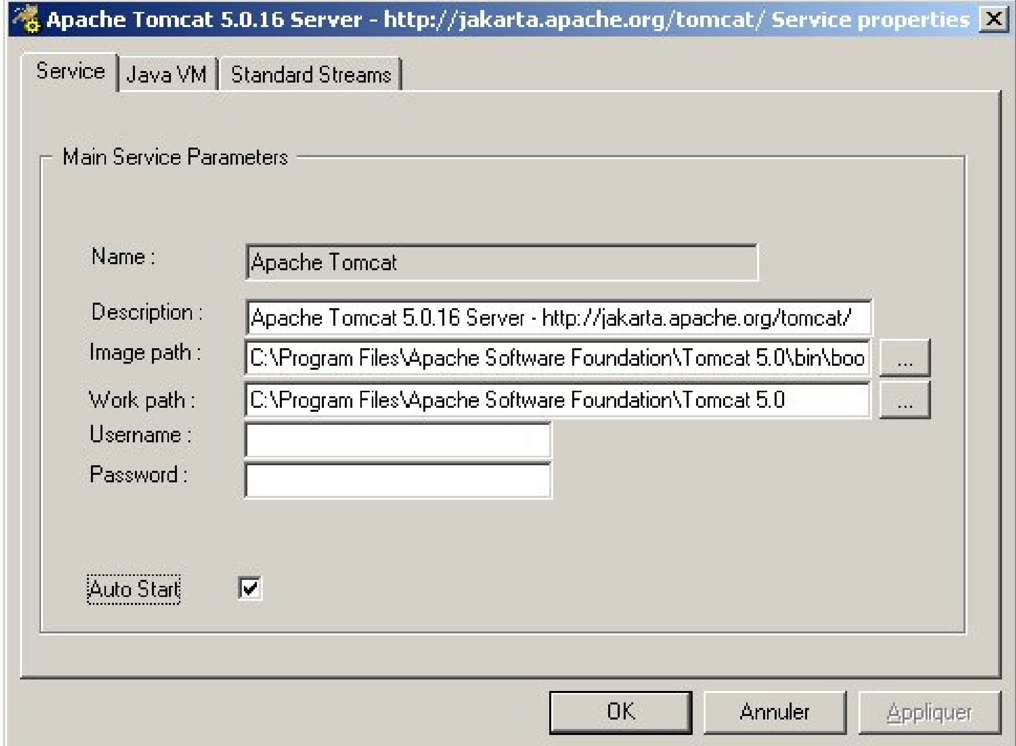
Elle possède un menu contextuel qui permet de réaliser plusieurs actions :

Pour arrêter Tomcat lorsqu'il est démarré de cette façon, il faut cliquer sur le bouton « Stop » dans l'onglet « General »

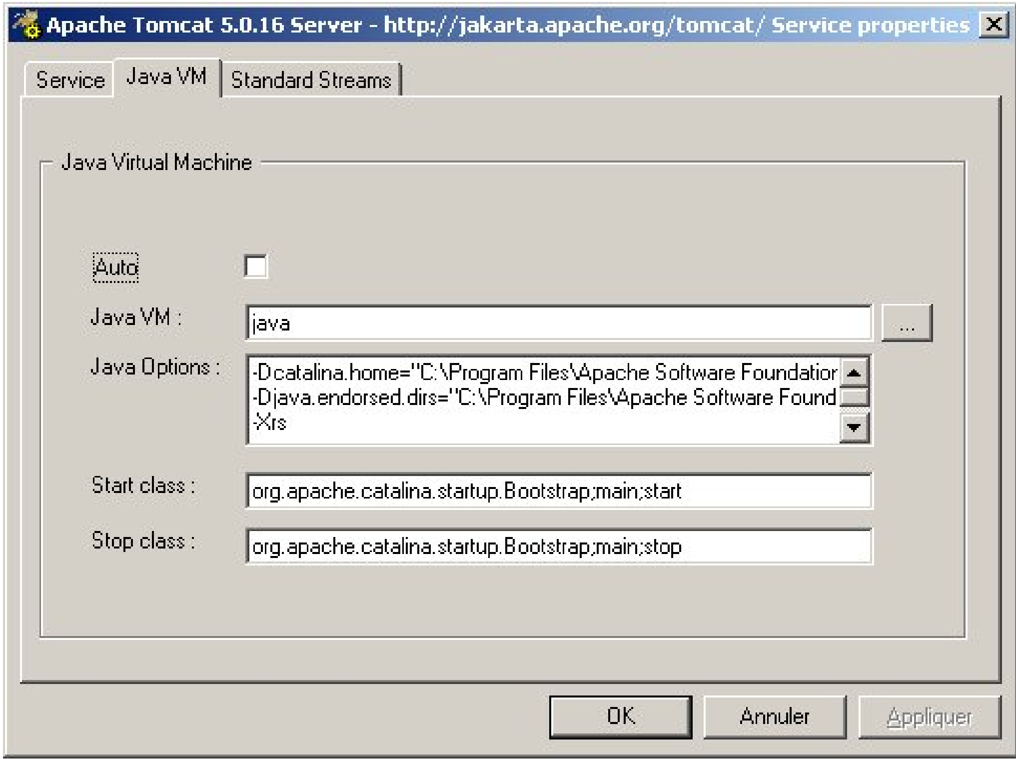
* « Open Console Monitor » : permet d'afficher un journal des messages émis par Tomcat



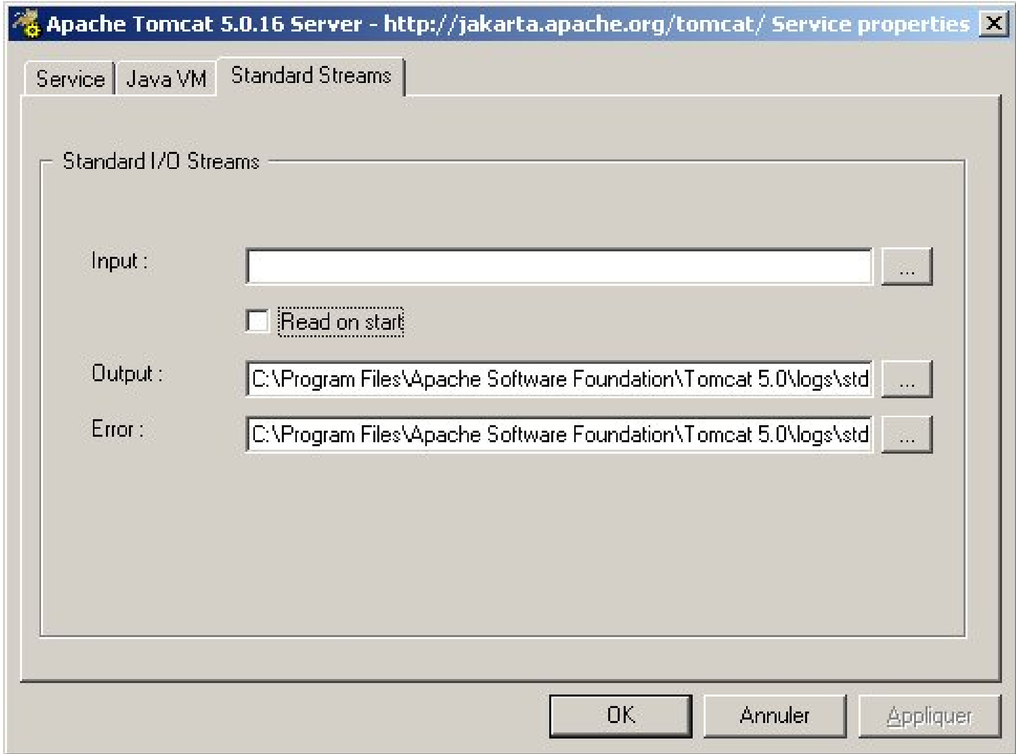
* « About » : permet d'afficher la licence de Tomcat
* « Properties » : permet de changer les propriétés de Tomcat
* L'onglet « Service » permet de préciser les informations générales concernant l'exécution de Tomcat



* L'onglet Java VM permet de préciser les options utilisées lors du lancement de la JVM dans laquelle Tomcat s'exécute



* L'onglet « Standard Streams » permet de préciser la localisation des fichiers de logs

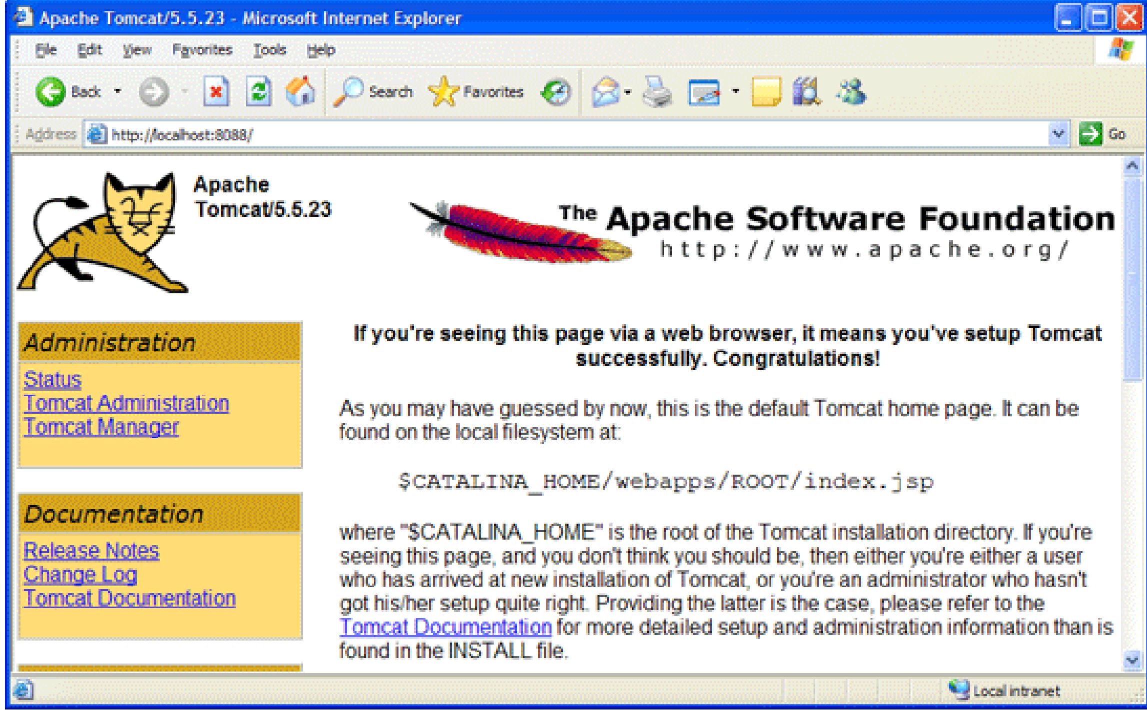


* « Shutdown » : permet d'arrêter le service Tomcat

71.3.3. La vérification de l'exécution

Pour vérifier la bonne exécution du serveur, il suffit d'ouvrir un navigateur et de saisir dans une url la machine hôte et le port d'écoute du connecteur http de Tomcat

Exemple :



Si Tomcat ne démarre pas :

- consulter les logs pour déterminer l'origine du problème.
- lancer Tomcat en utilisant le script Startup.bat dans une boîte de console Dos pour avoir afficher les logs.
- vérifier que le port utilisé n'est déjà utiliser par un autre service ou serveur

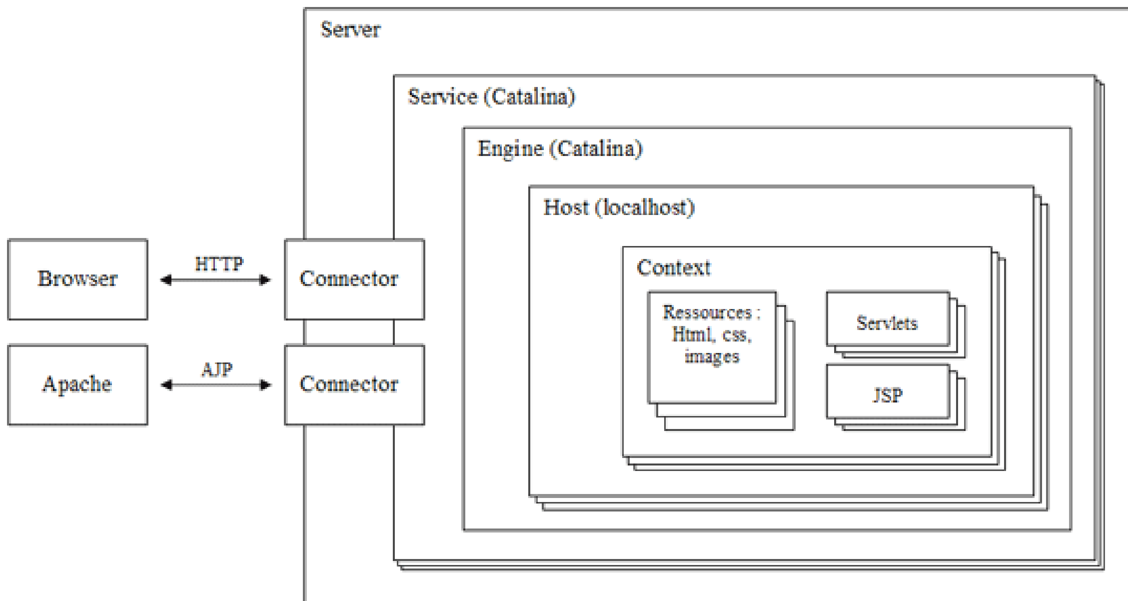
Si Tomcat est lancé mais que la page d'accueil ne s'affiche pas dans le navigateur :

- il faut vérifier l'url saisie (nom de l'hôte et surtout le numéro du port qui doit correspondre à celui configuré dans le fichier server.xml).
- si un proxy est utilisé, inhiber l'utilisation de se dernier pour l'url utilisée notamment en local

71.4. L'architecture

L'architecture de Tomcat est composée de plusieurs éléments :

- Server
  Le server encapsule tout le conteneur web. Il ne peut s'exécuter qu'un seul Server dans une JVM
- Service
  Un service regroupe des connectors et un unique engine
- Connector
  Un connector gère les communications avec un client. Tomcat propose plusieurs connecteurs notamment Coyote pour les communications par le protocole http, JK2 pour les communications par le protocole AJP
- Engine
  Un Engine traite les requêtes des différents Connector associés au Service : c'est le moteur de traitements des servlets.
- Host
  Un Host est un nom de domaine dont les requêtes sont traitées par Tomcat. Un Engine peut contenir plusieurs Host.
- Context
  Un context permet l'association d'une application web à un chemin unique pour un Host. Un Host peut avoir plusieurs contexts



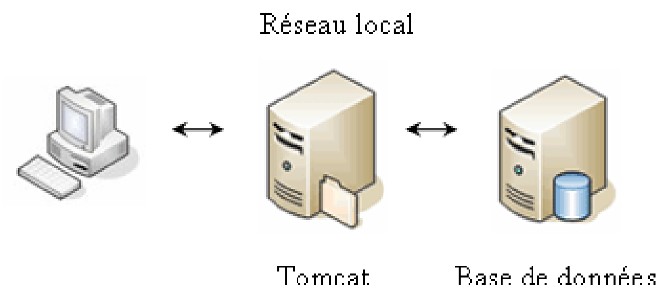
Pour assurer ces fonctionnalités, Tomcat utilise aussi différents types de composants qui prennent en charge des fonctionnalités particulières :

- Valve
  Une valve est une unité de traitements qui est utilisé lors du traitement de la requête. Leur rôle est similaire à celui des filtres pour les servlets.
- Logger
  Un Logger assure la journalisation des événements des différents éléments
- Realm
  Un Realm assure l'authentification et les habilitations pour un Engine

71.4.1. Les connecteurs

Qu'il soit utilisé en standalone ou en association avec un serveur web, Tomcat doit communiquer avec le monde extérieur.

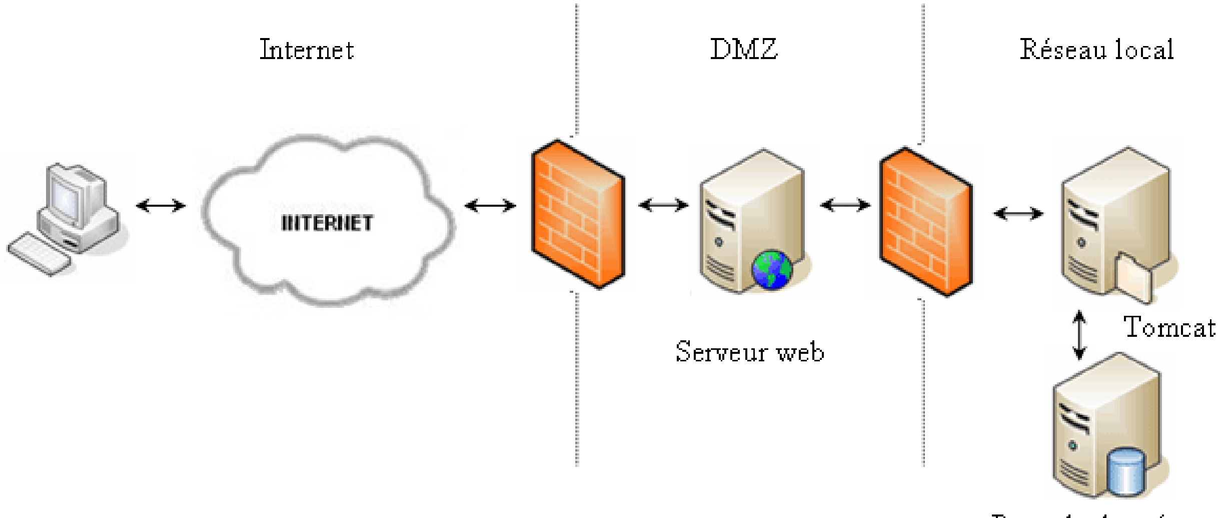
En mode Standalone, le connecteur mis en oeuvre utilise HTTP ou HTTPs pour communiquer.



En association avec un serveur web (Apache par exemple), ce dernier s'occupe des ressources statiques (page HTML, CSS, javascript, images,...) et Tomcat s'occupe des ressources dynamiques (JSP, servlets, ...).

Généralement pour des applications à usage externe, l'utilisation d'un serveur web et de Tomcat se fait dans une architecture réseau sécurisé grâce à une DMZ.

Exemple :



Le module mod_jk est utilisé pour assurer la communication entre le serveur Web (Apache par exemple) et Tomcat en utilisant le protocole AJP13.

71.4.2. Les services

Un service regroupe des connecteurs et l'engine. Par défaut Tomcat propose un seul service nommé Catalina.

Plusieurs services peuvent être définis dans un serveur Tomcat.

71.5. La configuration

La configuration de Tomcat est stockée dans plusieurs fichiers dans le sous répertoire conf. Le fichier de configuration principal est le fichier server.xml.

71.5.1. Le fichier server.xml

Tomcat est configuré grâce à un fichier xml nommé server.xml dans le répertoire conf.

La structure du document xml contenu dans le fichier server.xml reprend la structure de l'architecture de Tomcat :

Exemple :

```
<Server>
 <GlobalNamingResources/>
 <Service>
  <Connector/>
  <Engine>
   <Host/>
  </Engine>
 </Service>
</Server>
```

Remarque : il n'est pas possible d'utiliser un fichier server.xml de Tomcat 4 dans Tomcat 5.

71.5.1.1. Le fichier server.xml avec Tomcat 5

Le tag <Server> est le tag racine du fichier server.xml : il encapsule le serveur Tomcat lui-même. Il possède plusieurs attributs :

Attribut	Rôle
port	port sur lequel Tomcat écoute pour son arrêt (par défaut le port 8005)
shutdown	message à envoyer sur le port pour demander l'arrêt de Tomcat (par défaut SHUTDOWN)

Remarque : Tomcat refuse toute connexion sur le port d'arrêt sauf celle issue de la machine locale (exemple : telnet localhost 8005)

Le tag <Server> peut avoir un unique tag fils <GlobalNamingResources>, au moins un tag fils <Service> et éventuellement plusieurs tags fils <Listener>

Le tag <GlobalNamingResources/>
Ce tag encapsule des déclarations de ressources JNDI globales au serveur.

Exemple :

```
<GlobalNamingResources>
 <Environment
  name="simpleValue"
  type="java.lang.Integer"
  value="30"/>
<Resource
 auth="Container"
 description="User database that can be updated and saved"
 name="UserDatabase"
  type="org.apache.catalina.UserDatabase"
  pathname="conf/tomcat-users.xml"
  factory="org.apache.catalina.users.MemoryUserDatabaseFactory"/>
<Resource
 name="jdbc/MonAppDS"
 type="javax.sql.DataSource"
 password=""
  driverClassName="org.apache.derby.jdbc.EmbeddedDriver"
 maxIdle="2"
 maxWait="5000"
 username="APP"
 url="jdbc:derby:C:/Program Files/Java/jdk1.6.0/db/MaBaseDeTest"
 maxActive="4"/>
</GlobalNamingResources>
```

Le tag <Service> encapsule un service. Ce tag possède un seul attribut :

Attribut	Rôle
name	obligatoire : nom identifiant le service (par défaut Catalina)

Plusieurs services peuvent être définis dans un server : dans ce cas, chaque service doit avoir un attribut non distinct.

Le tag <Service> doit voir un moins un tag fils <Connector> et un unique tag fils <Engine>.

Le tag <Connector/> encapsule un connecteur. Un Connector se charge des échanges entre un client et le serveur pour un protocole donné. Ce tag possède plusieurs attributs :

Attribut	Rôle
className	nom de la classe d'implémentation du connector

protocol	protocole utilisé par le connecteur http ou AJP (par défaut HTTP/1.1). Pour utiliser le protocole AJP, il faut utiliser la valeur AJP/1.3
port	port d'écoute du connector (par défaut 8080 pour le protocole http et 8009 pour le protocole AJP13
redirectPort	sur un connector http, précise le port vers lequel les requêtes HTTPS seront redirigées
minSpareThreads	
maxSpareThreads	
maxThreads	nombre maximum de threads lancés pour traiter les requêtes
secure	positionné à true pour utiliser le protocole HTTPS (par défaut false)
enableLookups	effectue une résolution de l'adresse IP en nom de domaine dans les logs (par défaut true)
proxyName	
proxyPort	
acceptCount	Nombre maximum de requêtes mise à en attente si aucun thread n'est libre
connectionTimeout	timeout en milliseconde durant lequel une requête peut rester sans être traitée
disableUploadTimeout	
address	adresse IP des requêtes à traiter

Le tag <Engine> encapsule le moteur de servlet. Ce tag possède plusieurs attributs :

Attribut	Rôle
name	obligatoire : nom identifiant le moteur (par défaut Catalina)
defaultHost	obligatoire : hôte utilisé par défaut si l'hôte de la requête n'est pas défini dans le serveur
jvmRoute	utilisé dans le cadre d'un cluster de serveur Tomcat pour échanger des informations notamment celles des sessions

Le tag <Engine> doit avoir au moins un tag fils <Host> et peut avoir un tag fils unique <Logger>, <Realm>, <Valve> et <DefaultContext>

Exemple :

```
<Engine defaultHost="localhost" name="Catalina">
 <Realm ... />
 <Host ... />
</Engine>
```

Le tag <Host> défini un hôte virtuel sur le serveur. Ce tag possède plusieurs attributs :

Attribut	Rôle
name	obligatoire : nom de l'hôte
appBase	obligatoire : chemin par défaut pour les webapps de l'hôte. Ce chemin peut être absolu ou relatif par rapport au répertoire d'installation de Tomcat
defaultHost	hôte utilisé par défaut si l'hôte de la requête n'est pas défini dans le serveur
autoDeploy	activer le déploiement automatique des webapps. True par défaut

Exemple :

```
<Host appBase="webapps" name="localhost" autoDeploy="false">
</Host>
```

Le tag <Context> défini un contexte d'application. Ce tag possède plusieurs attributs :

Attribut	Rôle
docBase	obligatoire : chemin du répertoire ou de l'archive de l'application. Ce chemin peut être absolu ou relatif par rapport à l'attribut appBase du Host
reloadable	demande le rechargement automatique des classes modifiées (par défaut false)

71.5.1.2. Les valves

Une valve est une unité de traitements qui est utilisée lors du traitement de la requête. Leur rôle est similaire à celui des filtres pour les servlets.

Une est déclarée dans le fichier de configuration grâce au tag <valve> qui peut pet être utilisé comme fils des tags <engine>, <host> et <context>.

Une valve se présente sous la forme d'une classe qui implémente l'interface org.apache.catalina.Valve. Cette classe est précisée dans l'attribut className

Tomcat propose plusieurs valves par défaut :

Valve	Rôle
org.apache.catalina.valves.AccessLogValve	Configuration des logs
org.apache.catalina.authenticator.SingleSignOn	Utilisation du SSO

71.5.2. La gestion des rôles

Par défaut dans Tomcat, les rôles sont définis dans le fichier tomcat-users.xml. Ce fichier permet de définir des rôles et de les associer à des utilisateurs.

La modification du fichier tomcat-users.xml peut se faire directement sur le fichier ou via l'application d'administration

Exemple :

```
<?xml version='1.0' encoding='utf-8'?>
<tomcat-users>
 <role rolename="tomcat"/>
 <role rolename="manager"/>
 <role rolename="test"/>
 <role rolename="admin"/>
 <user username="tomcat" password="tomcat" roles="tomcat"/>
 <user username="jm" password="jm" roles="test"/>
 <user username="admin" password="baron" roles="admin,manager"/>
</tomcat-users>
```

Remarque : pour utiliser l'application d'administration ou le manager, il est nécessaire de définir les utilisateurs admin et manager

71.6. L'outil Tomcat Administration Tool

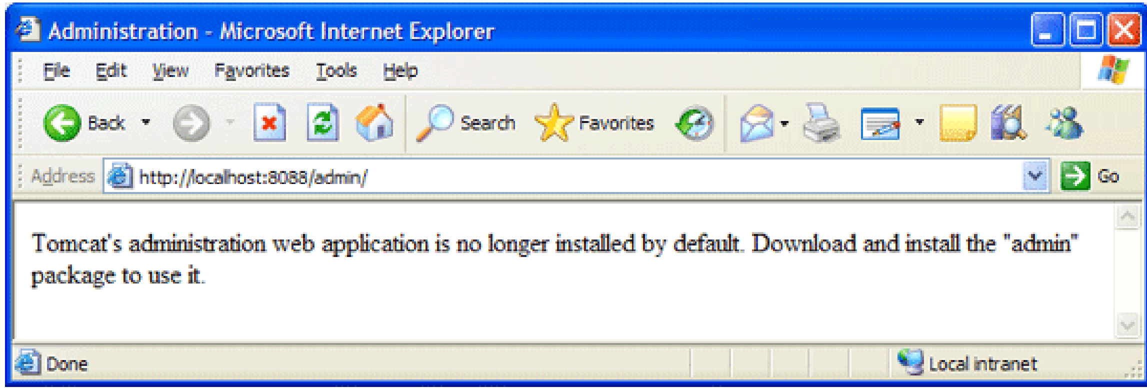
Tomcat Administration Tool est une application web qui permet de faciliter la configuration de Tomcat : il permet de modifier certains fichiers de configuration au moyen d'une interface graphique.

Remarque : cet outil n'est plus disponible à partir de Tomcat 6.

Il permet notamment de gérer :

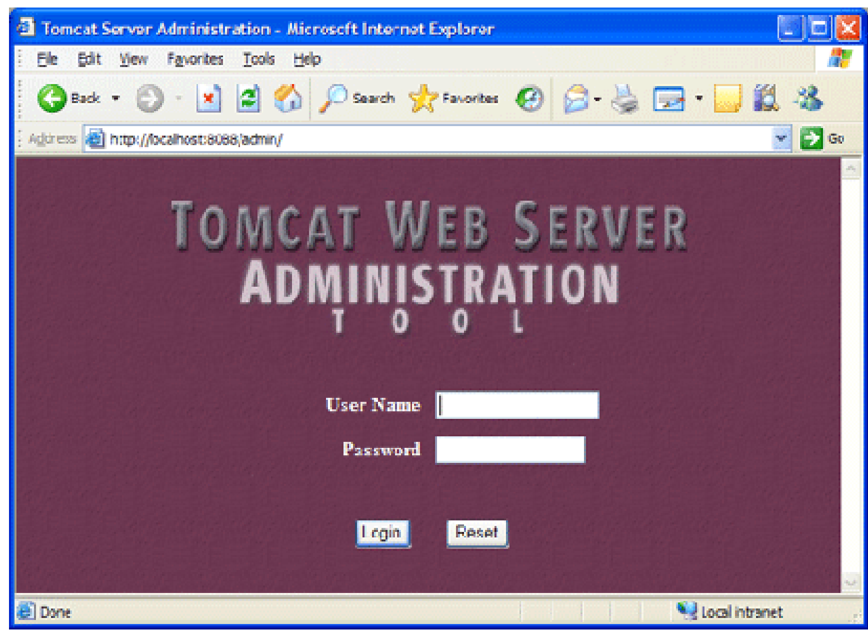
- Les connecteurs et les ports
- Les contextes d'applications
- Les ressources
- La sécurité
- Les utilisateurs

Remarque : à partir de Tomcat 5.5, cet outil n'est plus fourni en standard avec Tomcat et doit être téléchargé séparément et installé.

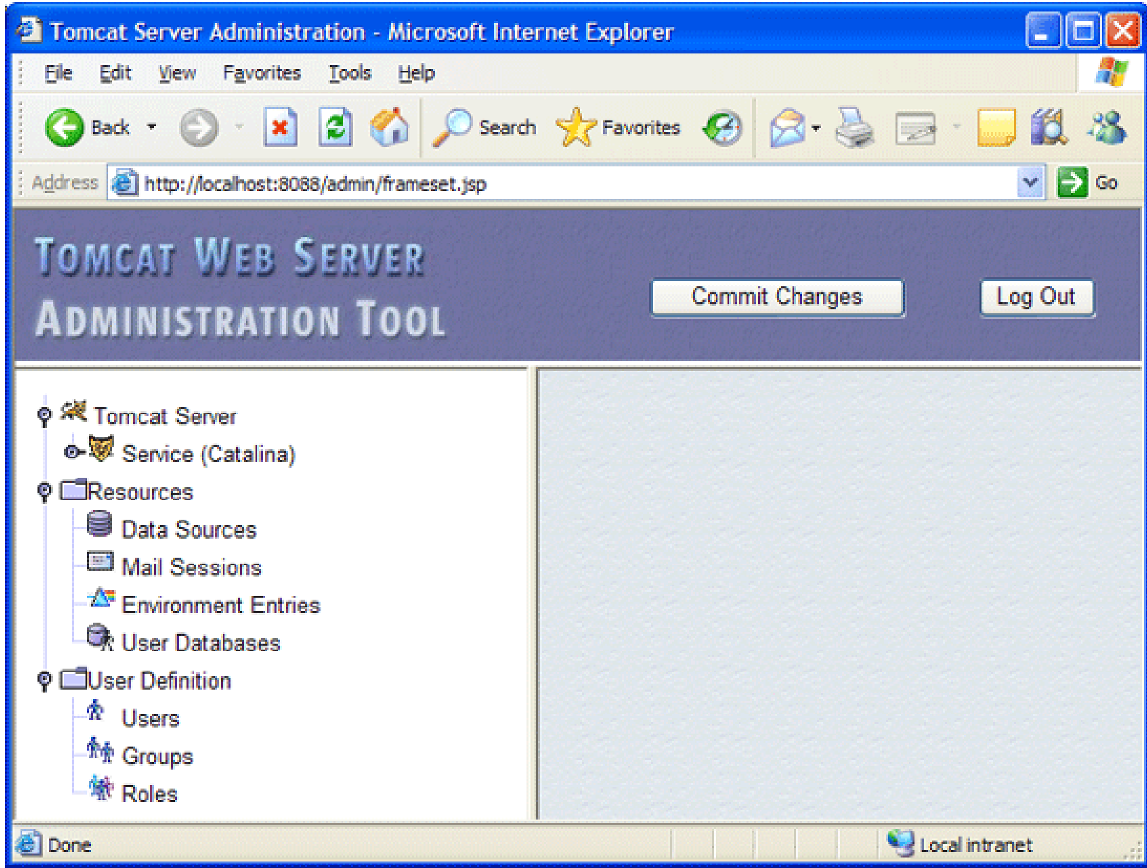


Dans ce cas, il faut télécharger le module et le décompresser dans le répertoire d'installation de Tomcat après avoir arrêté de serveur Tomcat.

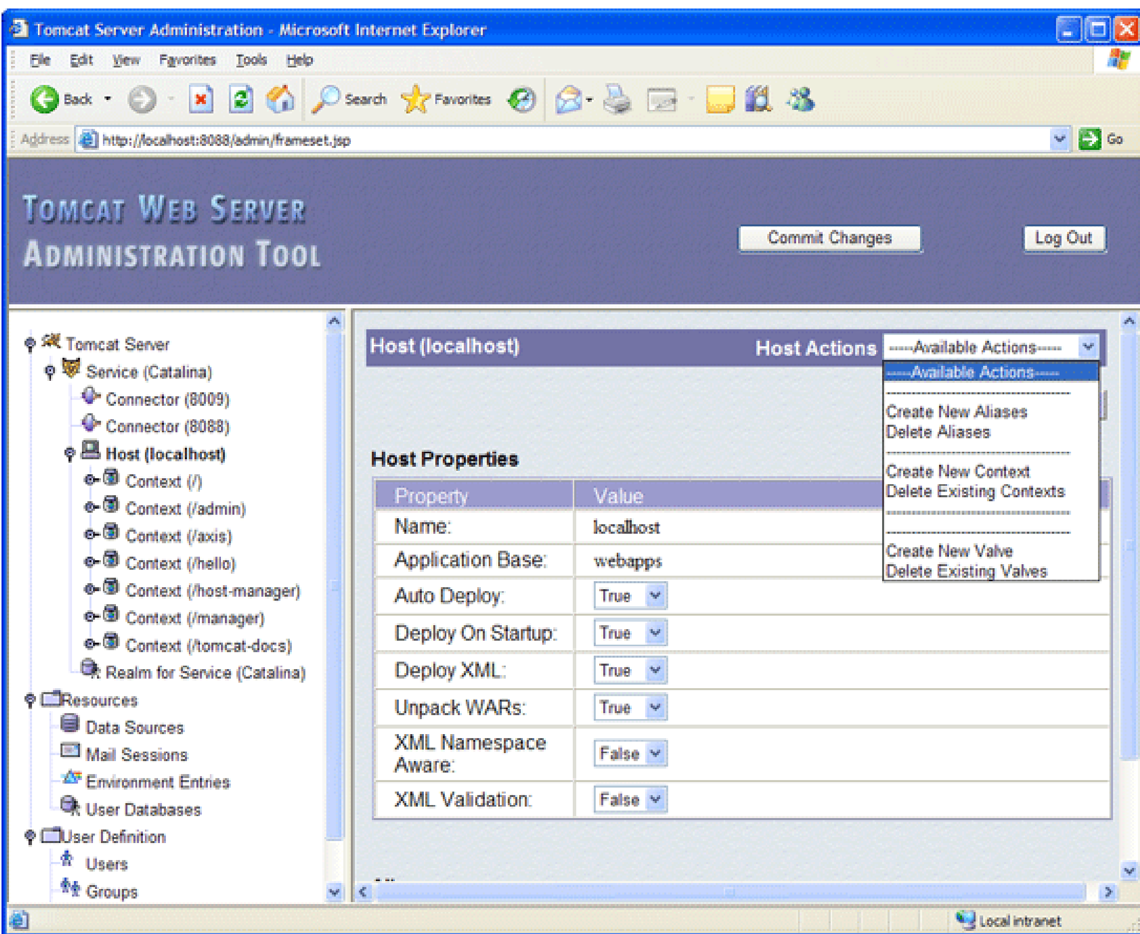
Pour lancer l'application, il suffit de cliquer sur le lien « Tomcat Administration » sur la page d'accueil de Tomcat



Il faut saisir le user et le mot de passe défini pour le user admin (information fournie au programme d'installation sous Windows)



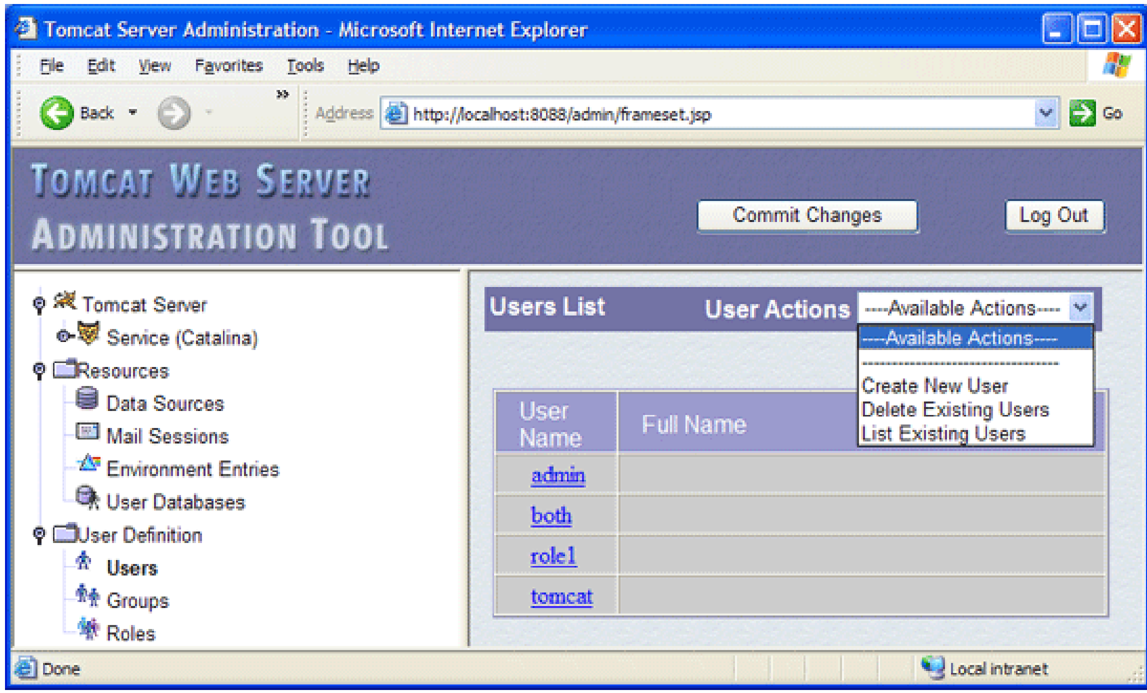
Les différents éléments configurables sont présentés sous une forme arborescente dans la partie de gauche. La partie de droite permet de modifier les données l'élément sélection. La liste déroulante « Actions » permet réaliser des actions en fonction du contexte (création d'élément, suppression, ...). Le bouton « Save » permet d'enregistrer les modifications en locale : il faut l'utiliser avant de changer de page.



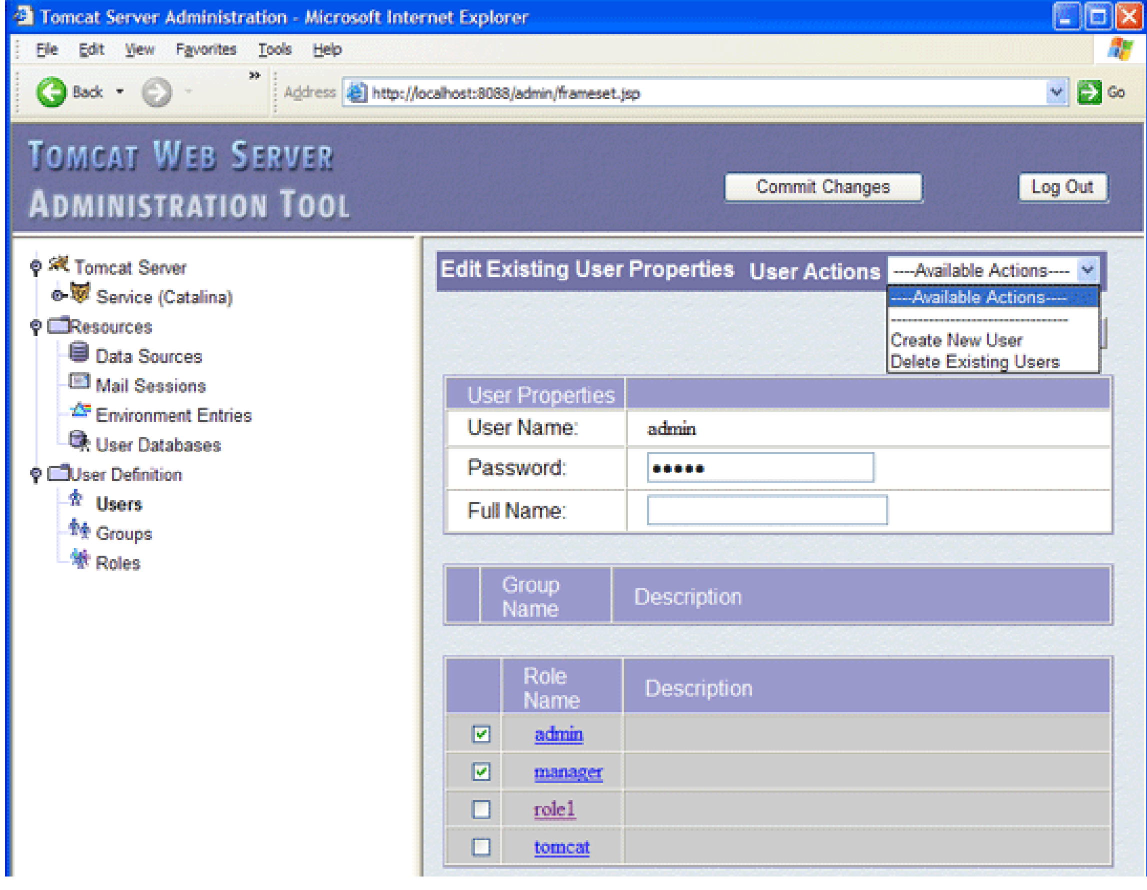
Attention : Il est très important pour valider les modifications de cliquer sur le bouton « Commit changes » : cet action rend persistantes les modifications en les écrivant dans les fichiers de configuration de Tomcat.

71.6.0.1. La gestion des utilisateurs, des rôles et des groupes

La partie « User Definition » de l'outil d'administration permet de gérer les users, les rôles et les groupes sans avoir à modifier directement le contenu du fichier tomcat-users.xml.

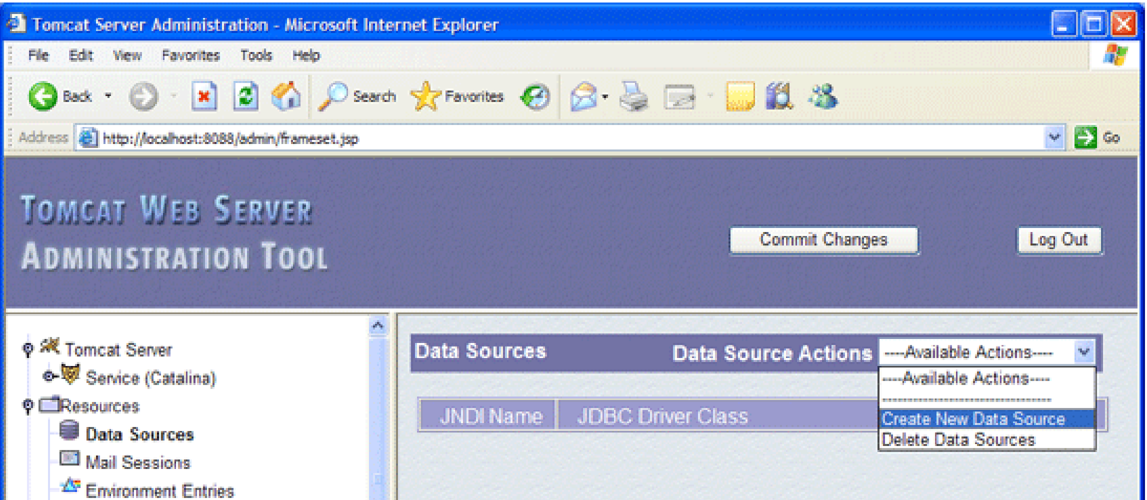


Pour modifier un user, il suffit de cliquer sur son lien.

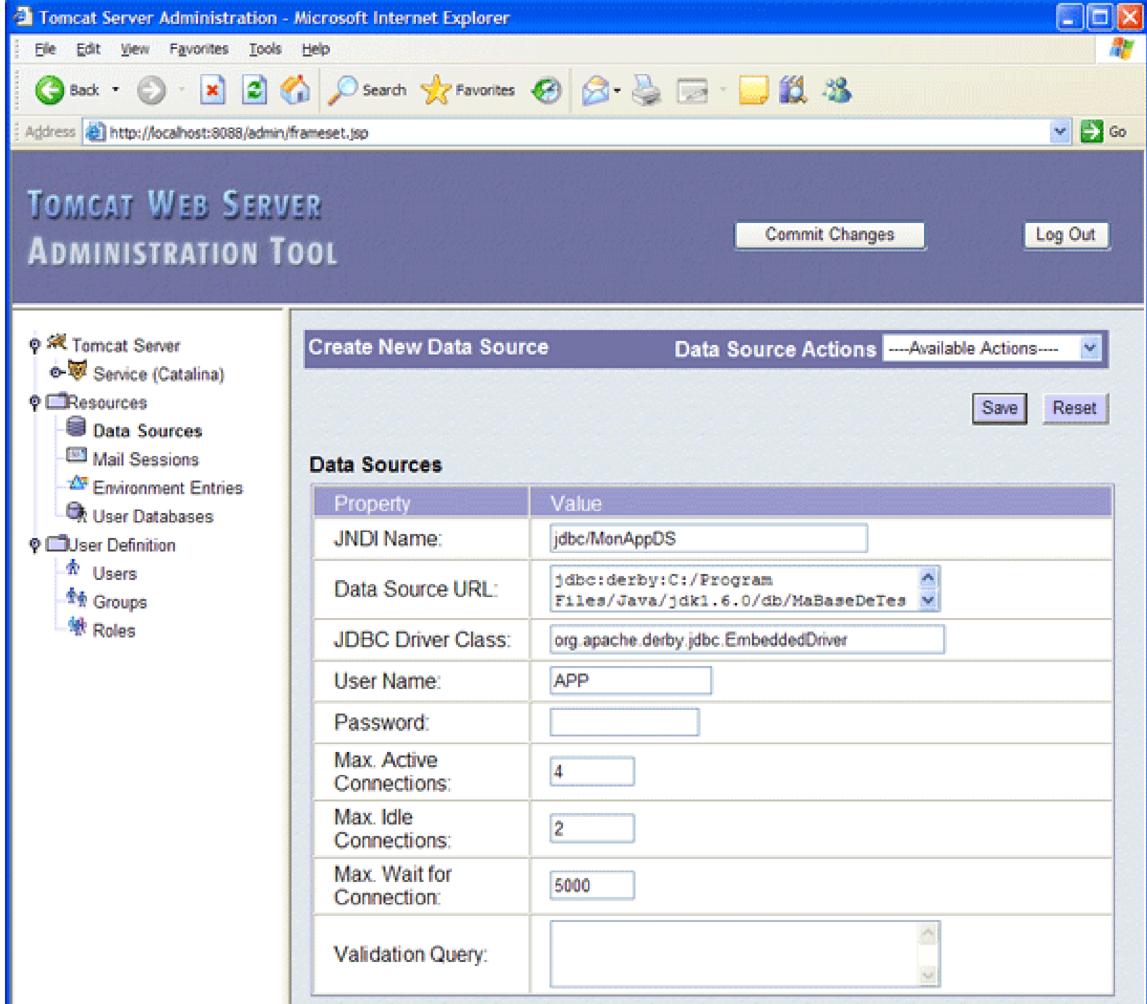


71.6.1. La création d'une DataSource dans Tomcat

Le plus simple est d'utiliser l'application d'administration



Cliquez sur « Data Sources » dans l'arborescence « Resources ». Dans la liste déroulante « DataSource Actions », sélectionnez « Create New Data Source »



Saisissez les informations nécessaire à la Datasource : le nom JNDI, l'url, la classe du pilote, le user, le mot de passe, ...

Cliquez sur le bouton « Save » puis sur « Commit changes » et confirmez la validation des modifications.

Dans le fichier de configuration de Tomcat server.xml, la Datasource a été ajoutée dans les ressources de GlobalNamingResources

Exemple :

```
<GlobalNamingResources>
...
 <Resource
  name="jdbc/MonAppDS"
  type="javax.sql.DataSource"
  password=""
   driverClassName="org.apache.derby.jdbc.EmbeddedDriver"
  maxIdle="2"
  maxWait="5000"
  username="APP"
  url="jdbc:derby:C:/Program Files/Java/jdk1.6.0/db/MaBaseDeTest"
  maxActive="4"/>
 </GlobalNamingResources>
```

71.7. Le déploiement des applications WEB

Pour être exécutée, une application web doit impérativement être déployée dans un conteneur de servlets même dans un environnement de développement.

Selon les spécifications des servlets depuis la version 2.2, un conteneur doit obligatoirement être capable de déployer une application web au format war. Tomcat propose aussi un support pour déployer les applications au format unpacked et propose différentes solutions pour assurer le déploiement des applications.

71.7.1. Déployer une application web avec Tomcat 5

Une application web peut être déployée sous Tomcat 5 de plusieurs manières :

- Copier l'application dans le répertoire webapps
- Définir un contexte
- Utiliser l'outil Tomcat Manager
- Utiliser les tâches Ant du Manager
- Utiliser l'outil TCD

71.7.1.1. Déployer une application au lancement de Tomcat

Alors que Tomcat est arrêté, il suffit de copier le répertoire contenant la webapp ou le fichier war qui la contient dans le sous répertoire webapps du répertoire de Tomcat et de redémarrer ce dernier.

Par défaut, l'uri de l'application utilisera le nom du répertoire ou du fichier war : Tomcat va créer un contexte pour l'application en lui associant comme chemin de contexte le nom du répertoire ou du fichier war sans son extension.

Par défaut, Tomcat décompresse le contenu d'un fichier war dans un répertoire portant le nom du fichier war sans son extension.

Pour redéployer une application sous la forme d'un fichier war, il est préférable de supprimer le répertoire contenant l'application décompressée.

Les applications du répertoire webapps sont automatiquement déployées au démarrage si l'attribut deployOnStartUp du tag Host vaut true.

71.7.1.2. Déployer une application sur Tomcat en cours d'exécution

Si l'attribut autoDeploy du tag Host vaut true, le déploiement de l'application par copie dans le répertoire webapps peut se faire alors que Tomcat est en cours d'exécution. Ce mécanisme permet aussi de recharger dynamiquement une application.

Remarque : Tomcat proposent des fonctionnalités de rechargement dynamique d'une application ayant subie des modifications : il est cependant préférable de redémarrer le serveur Tomcat pour éviter certains écueils.

71.7.1.3. L'utilisation d'un contexte

Un descripteur de contexte est un document au format xml qui contient la définition d'un contexte.

Ce descripteur permet de configurer le contexte

Ce fichier doit être placé dans le sous répertoire /conf/{engine_name}/{host_name} ou {engine_name} est le nom du moteur et {host_name} est le nom de l'hôte.

Avec la configuration par défaut de Tomcat, c'est le sous répertoire /conf/catalina/localhost

Le contenu de ce descripteur de contexte est détaillé dans une des sections suivantes de ce chapitre

71.7.1.4. Déployer une application avec le Tomcat Manager

Tomcat fourni l'application web Tomcat Manager pour permettre la gestion des applications web exécutée sur le serveur sans avoir à procéder à un arrêt/redémarrage de Tomcat.

Son utilisation est détaillée dans une des sections suivantes de ce chapitre.

71.7.1.5. Déployer une application avec les tâches Ant du Manager

Tomcat propose des tâches Ant qui permettent l'utilisation dans des scripts des certaines fonctionnalités du Manager.

L'utilisation de ces tâches Ant est détaillée dans la documentation de Tomcat.

71.7.1.6. Déployer une application avec le TCD

L'outil TCD (Tomcat Client Deployer) permet de packager une application et de gérer le cycle de vie de l'application, dans le serveur Tomcat. Cet outil utilise les tâches Ant du Manager.

Son utilisation est détaillée dans une des sections suivantes de ce chapitre.

71.8. Tomcat pour le développeur

71.8.1. Accéder à une ressource par son url

Une url permet d'accéder aux ressources statiques et dynamiques d'une application web. Par exemple dans une application contenue dans le sous répertoire maWebApp du répertoire webapps de Tomcat, l'accès aux ressources se fera avec une url de la forme :

http://host[:port]/webapp/chemin/ressource

Exemple :

http://localhost:8080/maWebApp

Pour une ressource statique, il suffit de préciser le chemin dans la webapp et le nom de la ressource.

Exemple : pour le fichier index.htm à la racine de la webapp

http://localhost:8080/maWebApp/index.htm

Exemple : pour le fichier index.htm dans le sous répertoire admin de la webapp

http://localhost:8080/maWebApp/admin/index.htm

Pour les ressources dynamiques de type servlet, le chemin et la ressource doivent correspondre au mapping qui est fait entre la classe et l'url dans le fichier de configuration web.xml

Exemple : mapping de la servlet com.jmdoudoux.mawebapp.AfficherListeServ vers l'url AfficherListe

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
        http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
  version="2.4">
 <servlet>
   <servlet-name>AffListe</servlet-name>
   <servlet-class>com.jmdoudoux.mawebapp.AfficherListeServ</servlet-class>
 </servlet>
 <servlet-mapping>
   <servlet-name>AffListe</servlet-name>
   <url-pattern>/AfficherListe</url-pattern>
 </servlet-mapping>
</web-app>
```

Exemple : appel de la servlet AfficherListeServ

http://localhost:8080/maWebApp/AfficherListe

Attention : Tomcat 5 vérifie le fichier de configuration de chaque webapp (WEB-INF/web.xml) et impose que l'ordre des tags dans ce fichier corresponde à celui défini dans la dtd du fichier web.xml.

71.8.2. La structure d'une application web et format war

Depuis la spécification 2.2 des servlets, le contenu d'une webapp doit obligatoirement respecter une certaine structure pour organiser ses répertoires et ses fichiers :

- Les ressources statiques sont stockées à la racine de la webapp ou dans un de ces sous répertoires autre que les répertoires WEB-INF et META-INF. Ces deux répertoires ne sont accessibles que par le conteneur
- Le fichier de configuration server.xml est stocké dans le répertoire WEB-INF
- Les ressources utilisés par le conteneur dans doivent être le répertoire WEB-INF et notamment son sous répertoire classes (pour les classes comme les servlets et les ressources associées comme les fichiers .properties) et son sous répertoire lib pour les bibliothèques (pilote JDBC, API, framework, ...)

Le format war est physiquement une archive de type zip qui englobe le contenu de la webapp.

Pour déployer une webapp dans Tomcat, il suffit de copier le répertoire de la webapp (forme unpacked) ou son fichier war (forme packed) dans le sous répertoire webapps.

Il est aussi possible de définir un contexte dont l'attribut docbase à pour valeur un répertoire quelconque du système de fichiers. Il est alors possible de développer l'application hors de Tomcat et d'utiliser ce répertoire de développement comme répertoire de déploiement.

Les classes et bibliothèques contenues dans les répertoires WEB-INF/classes et WEB-INF/lib sont utilisables par les classes de l'application.

71.8.3. La configuration d'un contexte

Un contexte est défini pour chaque application web exécutée sur le serveur soit explicitement dans la configuration soit implicitement avec un contexte par défaut créé par Tomcat.

71.8.3.1. La configuration d'un contexte avec Tomcat 5

Un contexte peut être défini explicitement de plusieurs manières :

- Soit dans le fichier conf/server.xml (depuis la version 5 de Tomcat, cette utilisation n'est pas recommandée)
- Soit un fichier xml de configuration du contexte
- Soit dans le fichier META-INF/context.xml de la webapp

Les contexts peuvent être modifiés manuellement en modification le fichier de configuration adéquat ou en utilisant l'outil d'administration de Tomcat

Un contexte est défini grâce à un tag <Context> qui possède plusieurs attributs :

- path : précise le chemin de contexte de l'application. Il doit commencer par un / et être unique pour chaque hôte
- docbase : précise le chemin absolu ou relatif par rapport au sous répertoire webapps du répertoire de l'application web ou le chemin de son fichier .war
- reloadable : précise si l'application doit être rechargée automatiquement en cas de modification dans les sous répertoires WEB-INF/Classes et WEB-INF/lib
- crossContext : précise si l'application peut avoir accès au contexte des autres applications exécutées en utilisant la méthode getContext() de la classe ServletContext (false par défaut).
- cookies : précise si les cookies sont utilisés pour échanger l'id de session (true par défaut). Pour forcer l'utilisation de la réécriture d'url, il faut mettre la valeur false à cet attribut
- privileged : précise si l'application peut avoir accès aux servlets du conteneur

L'implémentation par défaut de l'interface Context fournie avec Tomcat (***org.apache.catalina.core.StandardContext***) propose plusieurs attributs supplémentaires dont :

- workdir : précise le répertoire temporaire de travail
- unpackWar : précise si le fichier war doit être décompressé (true par défaut)

Un fichier de configuration du contexte peut être défini dans le répertoire conf/nom_engine/nom_hôte/. Son nom sera utilisé (dans son extension .xml) par défaut comme chemin de contexte.

La définition d'un context est par exemple utilisée par Sysdeo dans son plugin Eclipse pour faciliter l'utilisation de Tomcat.

71.8.4. L'invocation dynamique de servlets

Tomcat propose une fonctionnalité particulière nommé « Invoker servlet » qui permet l'appel d'une servlet sans que celle-ci soit déclarée dans un fichier web.xml.

Cette fonctionnalité peut être pratique dans un environnement de développement. Pour d'autres besoins que des tests, il ne faut pas utiliser cette fonctionnalité pour d'autres besoins et surtout elle ne doit pas être activée en production.

Pour activer cette fonctionnalité, il faut décommenter la déclaration de la servlet Invoker et son mapping dans le fichier de configuration par défaut des applications web. Ce fichier est le fichier /conf/web.xml.

La déclaration est faite par le tag

Exemple :

```
<servlet>
   <servlet-name>invoker</servlet-name>
  <servlet-class>
   org.apache.catalina.servlets.InvokerServlet
  </servlet-class>
  <init-param>
     <param-name>debug</param-name>
     <param-value>0</param-value>
  </init-param>
   <load-on-startup>2</load-on-startup>
</servlet>
```

Le mapping est fait par le tag

Exemple :

```
 <servlet-mapping>
   <servlet-name>invoker</servlet-name>
   <url-pattern>/servlet/*</url-pattern>
 </servlet-mapping>
```

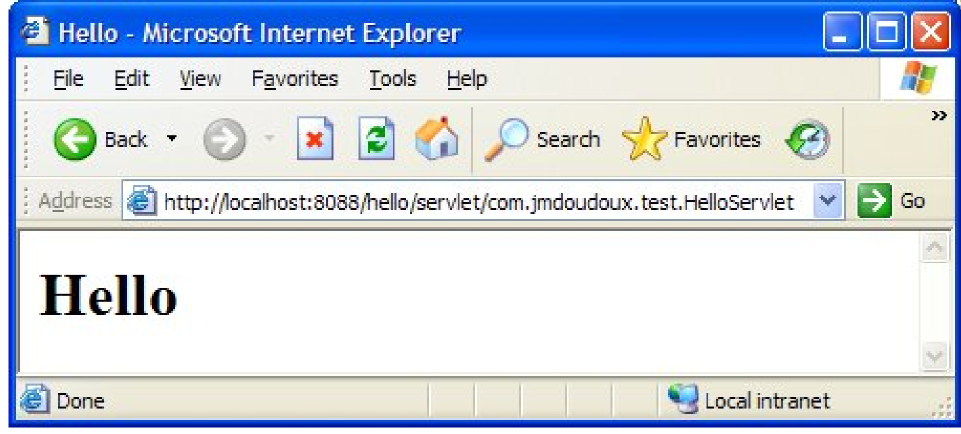
Il faut enregistrer le fichier modifié et redémarrer Tomcat.

Il suffit d'écrire le code de la servlet, de la compiler et de mettre le fichier .class correspondant dans le sous répertoire WEB-INF/Classes d'une webapp.

Exemple :

```
package com.jmdoudoux.test;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class HelloServlet extends javax.servlet.http.HttpServlet implements
   javax.servlet.Servlet {

  public HelloServlet() {
   super();
  }
  protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
   response.setContentType("text/html");
   PrintWriter out = response.getWriter();
   out.println("<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 "
     + "Transitional//EN\">\n"
     + "<HTML>\n"
     + "<HEAD><TITLE>Hello</TITLE></HEAD>\n"
     + "<BODY>\n"
     + "<H1>Hello</H1>\n"
     + "</BODY></HTML>");
  }
 }
```

Pour lancer la servlet, il suffit d'ouvrir l'url http://host:port/webapp/servlet/nom_pleinement_qualifié_de_la_classe



Attention : avec Tomcat 6, il est nécessaire de positionner l'attribut privileged à true pour le context de l'application.

71.8.5. Les bibliothèques partagées

Tomcat propose une solution pour partager des bibliothèques communes à toutes les applications qui s'exécutent sur le serveur.

Attention l'utilisation de cette fonctionnalité est spécifique à Tomcat. Il est en général préférable de mettre les bibliothèques dans le répertoire WEB-INF/lib de chaque application. Les bibliothèques sont dupliquées dans chaque application mais celle permet de rendre les applications moins dépendantes de Tomcat et chaque application pourrait utiliser une version différente de la bibliothèque.

71.8.5.1. Les bibliothèques partagées sous Tomcat 5

Deux répertoires sont fournis à cet effet :

- Le sous répertoire common/lib : les bibliothèques sont partagées par Tomcat et toutes les applications
- Le sous répertoire shared/lib : les bibliothèques sont partagées par toutes les applications mais sont pas accessible pour Tomcat
-

Des répertoires nommés classes permettent de façon similaire de partager des classes non regroupées dans une archive (jar ou zip).

Par défaut, Tomcat 5 fournis plusieurs bibliothèques partagées notamment celles des servlets, JSP et EL utilisable par toutes les webapp qu'il exécute. Ces API sont dans le répertoire "common/lib" ou "shared/lib" de Tomcat :

- ant.jar (Apache Ant 1.6)
- commons-collections*.jar (Commons Collections 2.1)
- commons-dbcp.jar (Commons DBCP 1.1)
- commons-el.jar (Commons Expression Language 1.0)
- commons-logging-api.jar (Commons Logging API 1.0.3)
- commons-pool.jar (Commons Pool 1.1)
- jasper-compiler.jar
- jsp-api.jar (JSP 2.0)
- commons-el.jar (JSP 2.0 EL)
- naming-common.jar
- naming-factory.jar
- naming-resources.jar
- servlet-api.jar (Servlet 2.4)

71.9. Le gestionnaire d'applications (Tomcat manager)

Tomcat fourni une application web pour permettre la gestion des applications web exécutée sur le serveur sans avoir à procéder à un arrêt/redémarrage de Tomcat.

Cette application permet :

- Lister les applications déployées avec leur état et le nombre de sessions ouvertes
- Déployer une nouvelle application (deploy)
- Arrêter (stop), démarrer (start) et recharger une application (reload)
- Supprimer une application (undeploy)
- Obtenir des informations sur la JVM et l'OS

L'application Manager peut être utilisée de trois manières :

- Utilisation de l'interface graphique est associée au contexte /manager. Elle peut être lancée en utilisant le lien « Tomcat Manager » sur la page d'accueil de Tomcat ou en utilisant l'uri /manager/html
- Utilisation de requêtes http : les opérations sont fournies dans la requête. Cette solution permet son utilisation dans des scripts
- Utilisation de tâches Ant

71.9.1. L'utilisation de l'interface graphique

L'utilisation du manager est soumise à une authentification préalable avec un utilisateur possédant le rôle manager. Ceci est configuré dans le fichier /conf/tomcat-users.xml.

Par défaut, aucun utilisateur ne possède ce rôle : il est donc nécessaire de l'ajouter.

Sous Windows, avec le programme d'installation, l'utilisateur saisi est associé au rôle admin et manager.

Par défaut, Tomcat utilise un MemoryRealm pour l'authentification Dans ce cas, pour ajouter ou modifier les utilisateurs et leurs rôles, il faut modifier le fichier /conf/tomcat-users.xml

Le tag <role> permet de définir un rôle. Il ajouter le rôle manager si ce dernier n'est pas défini.

Le tag <user> permet de déclarer un utilisateur en précisant son nom avec l'attribut username, son mot de passe avec l'attribut password et en lui associant un ou plusieurs rôles avec l'attribut roles. Plusieurs rôles peuvent être donnés à un utilisateur en les séparant par une virgule.

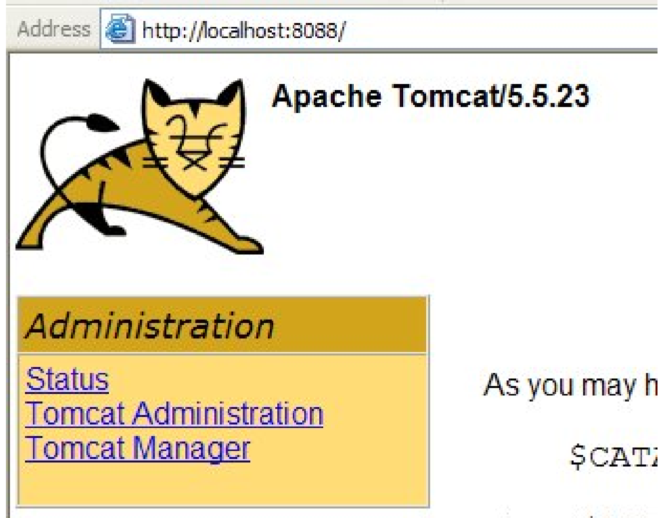
Exemple :

```
<?xml version='1.0' encoding='utf-8'?>
<tomcat-users>
 <role rolename="tomcat"/>
 <role rolename="role1"/>
 <role rolename="manager"/>
 <role rolename="admin"/>
 <user username="tomcat" password="tomcat" roles="tomcat"/>
 <user username="both" password="tomcat" roles="tomcat,role1"/>
 <user username="role1" password="tomcat" roles="role1"/>
 <user username="admin" password="admin" roles="admin,manager"/>
</tomcat-users>
```

Tous les utilisateurs qui possèdent le rôle manager peuvent être utilisés pour employer l'application Manager.

C'est une application web exécutée dans Tomcat qui permet de gérer les applications exécutée sous Tomcat. Elle est fournie en standard lors de l'installation de Tomcat.

Il faut ouvrir un navigateur sur l'url du serveur Tomcat.



Cliquez sur le lien « Tomcat Manager »

Une boîte de dialogue demande l'authentification d'un utilisateur ayant un rôle de type manager. Dans l'installation par défaut, le user admin possède les rôles manager et admin.

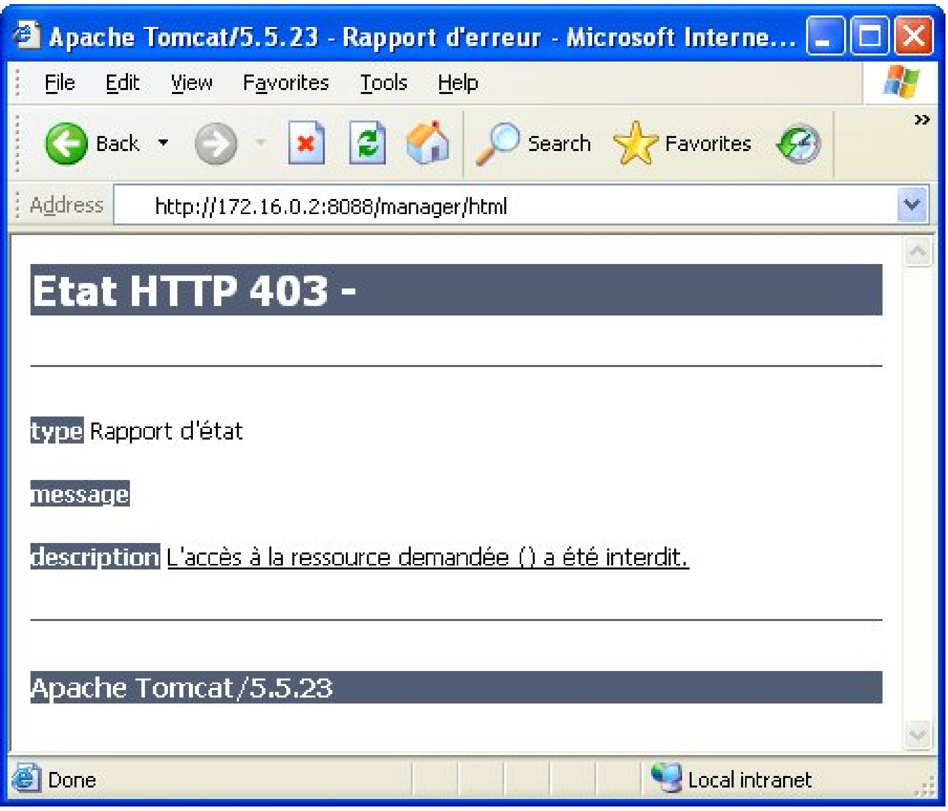
Remarque : les informations concernant les utilisateurs sont par défaut dans le répertoire $CATALINA_HOME/conf/tomcat-users.xml

Il est possible d'utiliser une valve pour restreindre l'accès à l'application Manager pour certaines machines en fonction de leur adresse IP ou de leur nom d'hôte

Exemple :

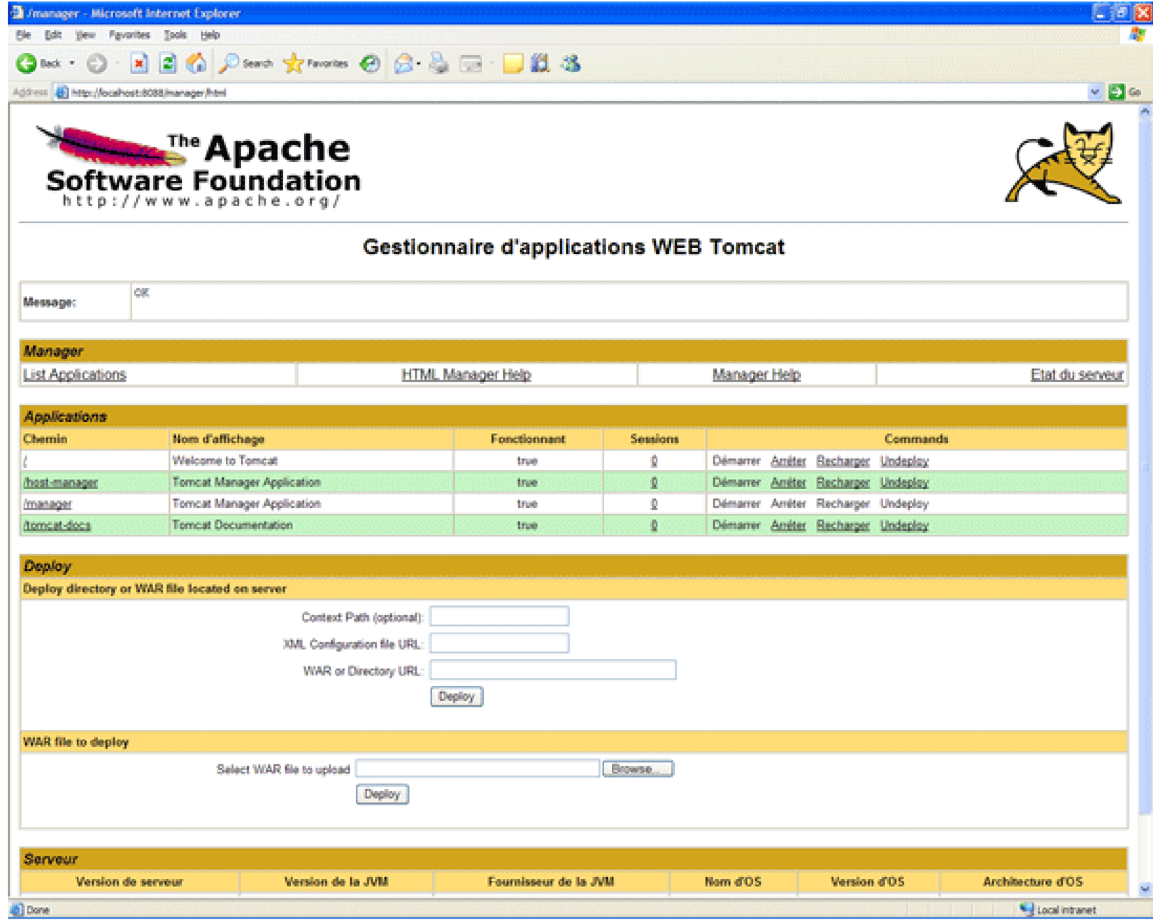
```
<?xml version="1.0" encoding="UTF-8"?>
<Context
  docBase="C:/Program Files/Apache/Tomcat 5.5/server/webapps/manager"
  privileged="true">
...
 <Valve className="org.apache.catalina.valves.RemoteAddrValve"
  allow="127.0.0.1"/>
</Context>
```

Si une machine non référencée tente d'accéder à l'application, un message d'erreur est affiché



Ceci permet de renforcer la sécurité notamment en production.

La page principale de l'application est composée de plusieurs parties.



La partie applications affiche la liste des applications déployées et permet de les gérer.

La partie Serveur affiche quelques informations sur les versions de Tomcat, de la JVM et de l'OS d'exécution.

La partie Deploy permet de déployer une application web soit à partir d'éléments sur le serveur soit sur le poste client.

71.9.1.1. Le déploiement d'une application

La partie « Deploy directory or WAR File located on Server » permet de déployer une application sur trouvant déjà sur la machine.

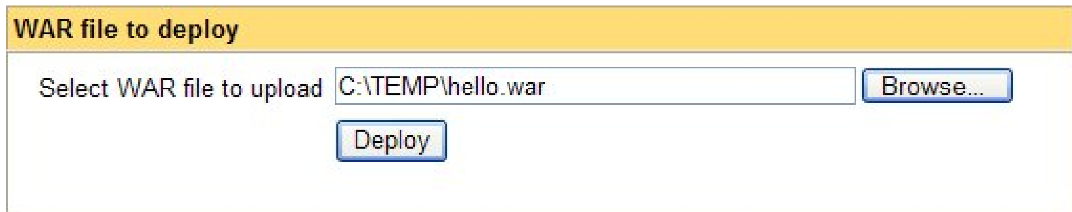
Il faut fournir trois données

- Le chemin du contexte : exemple : /MaWebApp
- Le nom du fichier web.xml : web.xml en général
- Le chemin ou se situe l'application : exemple : C:\java\projet\MaWebApp

Puis cliquer sur le bouton « Deploy »

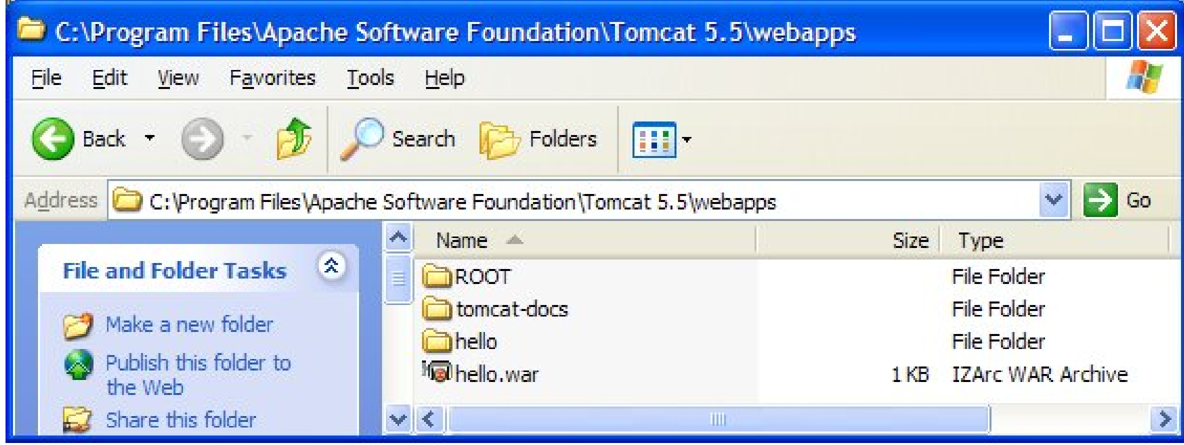
La partie « War file to deploy » permet de sélectionner un fichier de type war du poste client, de le télécharger sur le serveur, de le déployer dans Tomcat et de démarrer l'application.

Exemple :

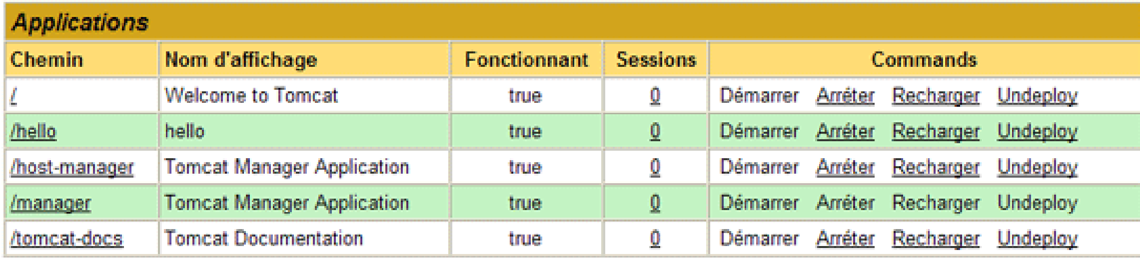


Cliquez sur « Browse », sélectionner le fichier .war et cliquez sur « Deploy »

Le fichier war est téléchargé dans le répertoire webapp, il est déployé par Tomcat (Tomcat est configuré par défaut pour déployer automatiquement les fichiers .war du répertoire webapp)



La liste des applications est enrichie de l'application déployée.



Applications				
Chemin	Nom d'affichage	Fonctionnant	Sessions	Commands
/	Welcome to Tomcat	true	0	Démarrer Arrêter Recharger Undeploy
/hello	hello	true	0	Démarrer Arrêter Recharger Undeploy
/host-manager	Tomcat Manager Application	true	0	Démarrer Arrêter Recharger Undeploy
/manager	Tomcat Manager Application	true	0	Démarrer Arrêter Recharger Undeploy
/tomcat-docs	Tomcat Documentation	true	0	Démarrer Arrêter Recharger Undeploy

71.9.1.2. La gestion des applications

La partie applications permet de gérer le cycle de vie des applications déployées.

Il est possible d'accéder à l'application en cliquant sur le lien du chemin de l'application

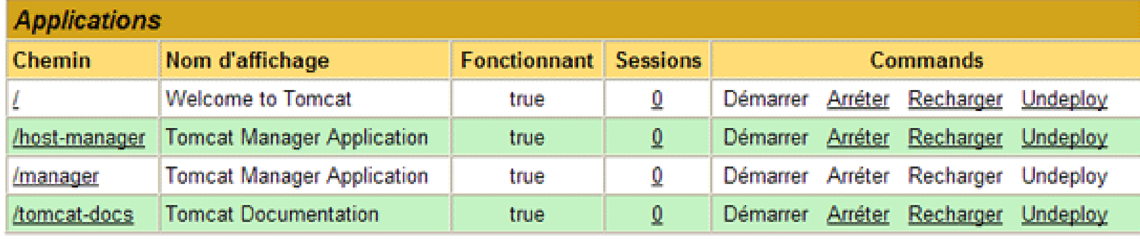
Il est possible d'arrêter l'application en cliquant sur le lien « Arrêter » de l'application correspondante.

Il est possible de démarrer l'application en cliquant sur le lien « Démarrer » de l'application correspondante.

Il est possible de recharger l'application (équivalent à un arrêt/démarrage consécutif) en cliquant sur le lien « Recharger » de l'application correspondante.

Il est possible de supprimer l'application en cliquant sur le lien « Undeploy » de l'application correspondante.

L'application est arrêter et supprimée du serveur Tomcat.



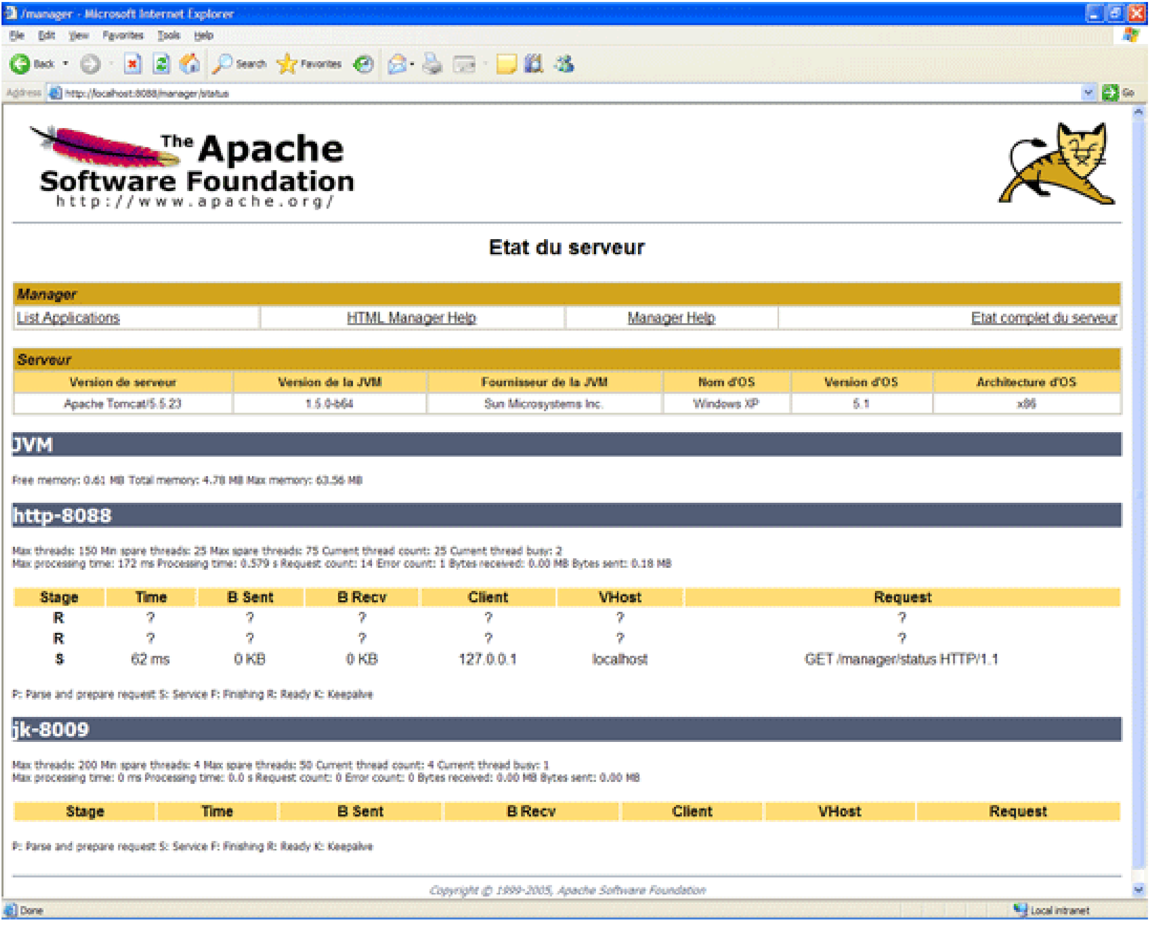
Applications

Chemin	Nom d'affichage	Fonctionnant	Sessions	Commands			
/	Welcome to Tomcat	true	0	Démarrer	Arrêter	Recharger	Undeploy
/host-manager	Tomcat Manager Application	true	0	Démarrer	Arrêter	Recharger	Undeploy
/manager	Tomcat Manager Application	true	0	Démarrer	Arrêter	Recharger	Undeploy
/tomcat-docs	Tomcat Documentation	true	0	Démarrer	Arrêter	Recharger	Undeploy

Chacune de ces actions nécessite une confirmation

Cliquez sur « Etat complet du serveur » pour obtenir des informations sur l'environnement d'exécution :

- Version de Tomcat,
- Information sur la JVM (version et fournisseur),
- Informations sur le système d'exploitation et l'architecture processeur,
- Informations sur la mémoire utilisée et disponible de la JVM,
- Statistiques des ports utilisés par Tomcat



71.9.2. L'utilisation des commandes par requêtes HTTP

Comme pour l'utilisation de l'interface, l'utilisation des commandes par requêtes http nécessite une authentification préalable.

Toutes les requêtes pour exécuter une commande sont de la forme :
http://{hôte}:{port}/manager/{commande}?{paramètres}

Hôte et port représente la machine et le port utilisé par Tomcat. Commande est la commande à exécuter avec ces éventuels paramètres.

Certaines commandes attendent un paramètre path qui précise le chemin du contexte de l'application à utiliser. La valeur de ce paramètre commence par un /.

Remarque : il n'est pas possible d'effectuer des commandes sur l'application Manager elle-même.

L'exécution de ces commandes renvoie une réponse ayant pour type mime text/plain. Cette réponse ne contient donc aucun tag de formatage HTML ce qui permet de l'exploiter dans des scripts par exemple.

La première ligne indique l'état de l'exécution de la commande : OK ou FAIL pour indiquer respectivement que la commande a réussi ou qu'elle a échoué. Le reste de la ligne contient un message d'information ou d'erreur.

Certaine commande renvoie des lignes supplémentaires contenant le résultat de la leur exécution.

71.9.2.1. La commande list

La commande list permet de demander l'affichage de la liste des applications déployées sur le serveur

Exemple : http://localhost:8088/manager/list

```
OK - Applications listées pour l'hôte virtuel (virtual host) localhost
/admin:running:0:C:/Program Files/Apache/Tomcat 5.5/server/webapps/admin
/host-manager:running:0:C:/Program Files/Apache/Tomcat 5.5/server/webapps/Manager
/tomcat-docs:running:0:tomcat-docs
/axis:running:0:axis
/:running:0:ROOT
/manager:running:0:C:/Program Files/Apache/Tomcat 5.5/server/webapps/manager
```

Cette liste contient plusieurs informations séparées par un point virgule : le contexte de l'application, son statut (running ou stopped), le nombre de session ouverte et le chemin de la webapp

71.9.2.2. La commande serverinfo

La commande serverinfo permet d'obtenir des informations sur l'OS et la JVM

Exemple : http://localhost:8088/manager/serverinfo

```
OK - Server info
Tomcat Version: Apache Tomcat/5.5.23
OS Name: Windows XP
OS Version: 5.1
OS Architecture: x86
JVM Version: 1.5.0-b64
JVM Vendor: Sun Microsystems Inc.
```

71.9.2.3. La commande reload

Cette commande permet de demander le rechargement d'une webapp qui est stockée dans un sous répertoire (déploiement sous la forme étendue).

Cette commande attend un paramètre path qui doit avoir comme valeur le contexte de la webapp

Exemple : http://localhost:8080/manager/reload?path=/hello

```
OK - Application rechargée au chemin de contexte /hello
Attention : le rechargement en concerne que les classes. Le fichier web.xml n'est pas relu :
la prise en compte de modification dans ce fichier nécessite un arrêt/démarrage de la webapp
```

71.9.2.4. La commande resources

La commandes resources permet d'obtenir une liste des ressources JNDI globales définie dans le serveur Tomcat et pouvant être utilisé.

Exemple : http://localhost:8080/manager/resources

```
OK - Liste des ressources globales de tout type
UserDatabase:org.apache.catalina.users.MemoryUserDatabase
jdbc/MonAppDS:org.apache.tomcat.dbcp.dbcp.BasicDataSource
simpleValue:java.lang.Integer
```

Chaque ressource est précisée sur une ligne qui contient le nom de la ressource et son type séparé par un deux points.

Il est possible de préciser un type d'objet grâce au paramètre type. Dans ce cas la valeur du paramètre type doit être une classe pleinement qualifiée.

Exemple : http://localhost:8088/manager/resources?type=java.lang.Integer

```
OK - Liste des ressources globales de type java.lang.Integer
simpleValue:java.lang.Integer
```

71.9.2.5. La commande roles

Cette commande donne la liste de tous les rôles définis

Exemple : http://localhost:8080/manager/roles

```
OK - Liste de rôles de securité
tomcat:
role1:
manager:
admin:
```

Chaque ligne contient un rôle et sa description séparée par un caractère deux points.

71.9.2.6. La commande sessions

Cette commande permet d'obtenir des informations sur les sessions d'un contexte.

Cette commande attend obligatoirement le paramètre path qui précise le chemin du contexte de l'application. Si ce paramètre n'est pas précisé, la commande renvoie une erreur.

Exemple : http://localhost:8080/manager/sessions

```
ECHEC - Un chemin de contexte invalide null a été spécifié
```

Le résultat de la commande contient :

- Le timeout des sessions
- Le nombre de sessions actives par tranche de timeout de 10 minutes

Exemple : http://localhost:8088/manager/sessions?path=/hello

```
OK - Information de session pour l'application au chemin de contexte /hello
Interval par défaut de maximum de session inactive 30 minutes
30 - <40 minutes:2 sessions
```

71.9.2.7. La commande stop

Cette commande permet de demander l'arrêt d'une webapp.

Cette commande attend obligatoirement le paramètre path qui précise le chemin du contexte de l'application à arrêter. Si ce paramètre n'est pas précisé, la commande renvoie une erreur.

Exemple : http://localhost:8080/manager/stop

```
ECHEC - Un chemin de contexte invalide null a été spécifié
```

La commande renvoie son statut et un message d'information

Exemple : http://localhost:8080/manager/stop?path=/hello

```
OK - Application arrétée pour le chemin de contexte /hello
```

71.9.2.8. La commande start

Cette commande permet de démarrer une webapp.

Cette commande attend obligatoirement le paramètre path qui précise le chemin du contexte de l'application à démarrer. Si ce paramètre n'est pas précisé, la commande renvoie une erreur.

Exemple : http://localhost:8080/manager/start

```
ECHEC - Un chemin de contexte invalide null a été spécifié
```

La commande renvoie son statut et un message d'information

Exemple : http://localhost:8088/manager/start?path=/hello

```
OK - Application démarrée pour le chemin de contexte /hello
```

71.9.2.9. La commande undeploy

Cette commande permet de supprimer une webapp. Elle arrête préalablement l'application avant sa suppression.

Cette commande attend obligatoirement le paramètre path qui précise le chemin du contexte de l'application à démarrer. Si ce paramètre n'est pas précisé, la commande renvoie une erreur.

Exemple : http://localhost:8080/manager/undeploy

```
ECHEC - Un chemin de contexte invalide null a été spécifié
```

La commande renvoie son statut et un message d'information

Exemple : http://localhost:8080/manager/undeploy?path=/hello

```
OK - Application non-déployée pour le chemin de contexte /hello
```

Attention : cette commande supprime tout ce qui concerne la webapp

- Le répertoire de déploiement de l'application si il existe (par exemple webapps/hello)
- Le fichier .war si il existe (par exemple webapps/hello.war)
- Le contexte

Si le contexte est défini dans le fichier server.xml, alors la commande échoue

Exemple : le fichier server.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<Server>
...
 <Service name="Catalina">
...
  <Engine defaultHost="localhost" name="Catalina">
   <Realm className="org.apache.catalina.realm.UserDatabaseRealm"/>
   <Host appBase="webapps" name="localhost">
    <Context path="/hello"></Context>
   </Host>
  </Engine>
 </Service>
</Server>
```

Exemple : http://localhost:8088/manager/undeploy?path=/hello

```
FAIL - Context /hello is defined in server.xml and may not be undeployed
```

71.9.2.10. La commande deploy

Cette commande permet de déployer une application.

Exemple : déployer l'application dont le fichier .war est dans le répertoire de déploiement

```
http://localhost:8080/manager/deploy?path=/hello&war=hello.war
```

Remarque : cet exemple n'est utile que si l'option autoDeploy est à False dans la configuration du Host concerné.

Exemple : déployer une application dont le répertoire de déploiement est un sous répertoire du répertoire de déploiement de Tomcat

```
http://localhost:8088/manager/deploy?war=hello&path=/hello

OK - Application déployée pour le chemin de contexte /hello
```

Remarque : cet exemple n'est utile que si l'option autoDeploy est à False dans la configuration du Host concerné. Il permet de déployer une application sous la forme d'un sous répertoire dans le répertoire de déploiement de Tomcat sans avoir à redémarrer Tomcat.

71.9.3. L'utilisation du manager avec des tâches Ant

Tomcat 5 propose un ensemble de tâches Ant qui permet d'exécuter des traitements du manager.

Comme pour toute tâche Ant externe, il faut déclarer chaque tâche à utiliser avec le tag taskdef.

Exemple :

```
<project name="Hello" default="list" basedir=".">
 <!-- Propriété d'accès au Manager -->
 <property name="url" value="http://localhost:8088/manager" />
 <property name="username" value="admin" />
 <property name="password" value="admin" />
 <!-- Chemin du contexte de l'application -->
 <property name="path" value="/hello" />
 <!-- Configure the custom Ant tasks for the Manager application -->
 <taskdef name="list" classname="org.apache.catalina.ant.ListTask" />
 <target name="list" description="Liste des webapp déployée">
  <list url="${url}" username="${username}" password="${password}" />
```

```
 </target>
</project>
```

Pour utiliser les tâches, il faut que le fichier catalina-ant.jar soit accessible par Ant. Pour cela, il y a deux solutions :
1) Dans la balise classpath de la balise taskdef

Exemple :

```
<project name="Hello" default="list" basedir=".">
 <property name="tomcat.home" value="C:/Program Files/Apache/Tomcat 5.5" />

 <!-- Propriété d'accès au Manager -->
 <property name="url" value="http://localhost:8088/manager" />
 <property name="username" value="admin" />
 <property name="password" value="admin" />

 <!-- Chemin du contexte de l'application -->
 <property name="path" value="/hello" />

 <!-- Configure the custom Ant tasks for the Manager application -->
  <taskdef name="list" classname="org.apache.catalina.ant.ListTask">
  <classpath>
   <path location="${tomcat.home}/server/lib/catalina-ant.jar" />
  </classpath>
 </taskdef>

 <target name="list" description="Liste des webapp déployée">
  <list url="${url}" username="${username}" password="${password}" />
 </target>
</project>
```

2) copier le fichier catalina-ant.jar contenu dans le sous répertoire server/lib du répertoire d'installation de Tomcat dans le sous répertoire lib du répertoire d'installation de Ant.

Résultat d'exécution :

```
Buildfile: C:\Documents and Settings\jmd\workspace\Tests\build.xml
list:
   [list] OK - Applications listées pour l'hôte virtuel (virtual host) localhost
   [list] /admin:running:0:C:/Program Files/Apache/Tomcat 5.5/server/webapps/admin
   [list] /host-manager:running:0:C:/Program Files/Apache/Tomcat 5.5/server/webapps/
            host-manager
   [list] /tomcat-docs:running:0:tomcat-docs
   [list] /hello:running:0:hello
   [list] /axis:running:0:axis
   [list] /:running:0:ROOT
   [list] /manager:running:0:C:/Program Files/Apache/Tomcat 5.5/server/webapps/manager
BUILD SUCCESSFUL
Total time: 173 milliseconds
```

Tomcat propose plusieurs tâches :

Classe de la tâche	Rôle
org.apache.catalina.ant.InstallTask	Déployer une application
org.apache.catalina.ant.ReloadTask	Recharger une application
org.apache.catalina.ant.ListTask	Liste des applications déployées
org.apache.catalina.ant.StartTask	Démarrer une application
org.apache.catalina.ant.StopTask	Arrêter une application
org.apache.catalina.ant.UndeployTask	Supprimer une application

org.apache.catalina.ant.SessionsTask	Obtenir des informations sur la session
org.apache.catalina.ant.RôlesTask	Obtenir des informations sur les rôles
org.apache.catalina.ant.ServerInfoTask	Obtenir des informations sur le serveur
org.apache.catalina.ant.ResourcesTask	Obtenir des informations sur les ressources JNDI
org.apache.catalina.ant.JMXGetTask	Obtenir une valeur d'un MBean
org.apache.catalina.ant.JMXQueryTask	Effectuer une requête sur le serveur JMX
org.apache.catalina.ant.JMXSetTask	Mettre à jour une valeur d'un MBean

Toutes ces tâches attendent au moins trois paramètres :

- url : url d'accès à l'application Manager
- username : utilisateur ayant le rôle admin
- password : mot de passe de l'utilisateur ayant le rôle admin

Certaines tâches attendent en plus des paramètres dédiés à leur exécution.

71.9.4. L'utilisation de la servlet JMXProxy

Tomcat propose une servlet qui fait office de proxy pour obtenir ou mettre à jour des données de MBean.

Par défaut, sans paramètre la servlet affiche tous les MBeans.

Exemple : http://localhost:8088/manager/jmxproxy

```
OK - Number of results: 200

Name: Users:type=Role,rolename=role1,database=UserDatabase
modelerType: org.apache.catalina.mbeans.RoleMBean
rolename: role1

Name: Users:type=User,username="both",database=UserDatabase
modelerType: org.apache.catalina.mbeans.UserMBean
groups: [Ljava.lang.String;@16be68f
password: tomcat
roles: [Ljava.lang.String;@edf389
username: both

Name: Catalina:type=Manager,path=/axis,host=localhost
modelerType: org.apache.catalina.session.StandardManager
algorithm: MD5
randomFile: /dev/urandom
className: org.apache.catalina.session.StandardManager
distributable: false
entropy: org.apache.catalina.session.StandardManager@a4e743
maxActiveSessions: -1
maxInactiveInterval: 1800
processExpiresFrequency: 6
sessionIdLength: 16
name: StandardManager
pathname: SESSIONS.ser
activeSessions: 0
sessionCounter: 0
maxActive: 0
sessionMaxAliveTime: 0
sessionAverageAliveTime: 0
rejectedSessions: 0
expiredSessions: 0
processingTime: 0
```

```
duplicates: 0
...
```

Le paramètre qry permet de préciser une requête pour filtrer les résultats

Exemple : http://localhost:8088/manager/jmxproxy/?qry=*%3Atype=User%2c*

```
OK - Number of results: 4

Name: Users:type=User,username="both",database=UserDatabase
modelerType: org.apache.catalina.mbeans.UserMBean
groups: [Ljava.lang.String;@1d8c528
password: tomcat
roles: [Ljava.lang.String;@77eaf8
username: both

Name: Users:type=User,username="tomcat",database=UserDatabase
modelerType: org.apache.catalina.mbeans.UserMBean
groups: [Ljava.lang.String;@e35bb7
password: tomcat
roles: [Ljava.lang.String;@9a8a68
username: tomcat

Name: Users:type=User,username="role1",database=UserDatabase
modelerType: org.apache.catalina.mbeans.UserMBean
groups: [Ljava.lang.String;@1f4e571
password: tomcat
roles: [Ljava.lang.String;@1038de7
username: role1

Name: Users:type=User,username="admin",database=UserDatabase
modelerType: org.apache.catalina.mbeans.UserMBean
groups: [Ljava.lang.String;@5976c2
password: admin
roles: [Ljava.lang.String;@183e7de
username: admin
```

La servlet permet aussi de modifier à chaud des attributs d'un MBean grâce à trois paramètres :

- set : le nom du MBean
- att : le nom de l'attribut à modifier
- val : la nouvelle valeur de l'attribut

71.10. L'outil Tomcat Client Deployer

L'outil TCD (Tomcat Client Deployer) permet de packager une application et de gérer le cycle de vie de l'application dans le serveur Tomcat. Cet outil repose sur les tâches Ant qui utilisent le Tomcat Manager

Il faut installer l'outil Ant :

- Télécharger Ant
- Décompresser le fichier obtenu
- Ajouter le répertoire bin d'Ant dans le path du système

La variable d'environnement JAVA_HOME doit pointer sur le répertoire d'un JDK

Il faut installer l'outil TDC en effectuant les opérations suivantes :

- Télécharger TDC
- Décompresser le fichier obtenu

Il faut définir un fichier deployer.properties qui va contenir des informations sur l'application à gérer et sur le serveur Tomcat. Ces informations sont fournies sous la forme de propriétés :

- webapp : nom de l'application web
- path : chemin de contexte de l'application
- url : url vers le manager de Tomcat
- username : nom de l'utilisateur ayant le rôle manager
- password : mot de passe de l'utilisateur
- build : répertoire de base dans lequel l'application sera compilée. L'application sera compilée dans le répertoire ${build}/webapp/${path}

Exemple :

```
webapp=hello
path=/hello
url=http://localhost:8088/manager
username=admin
password=admin
```

Pour exécuter TCD, il faut lancer ant avec en paramètre la tâche à exécuter dans le répertoire qui contient le fichier build.xml

Exemple :

```
Ant start
Ant stop
Ant reload
```

Les tâches utilisables sont :

- compile : compile et valide une application (classes et JSP). Cette tâche ne nécessite pas d'accès au serveur Tomcat. Attention : cette compilation ne peut fonctionne qu'avec la version de Tomcat correspond à celle de l'outil
- deploy : déploie une application dans le serveur Tomcat
- start : démarre l'application
- reload : rechargement de l'application
- stop : arrêt de l'application

71.11. Les optimisations

Cette section présente rapidement quelques optimisations possibles dans la configuration de Tomcat notamment dans une optique d'exécution dans un environnement de production.

Il est préférable de mettre à false l'attribut enableLookups des tags <Connector> dans le fichier server.xml : ceci évite à Tomcat de déterminer le nom de domaine à partir de l'adresse IP des requêtes

Il est préférable de remplacer les valeurs des attributs name des tags <Service> et <Engine> dans le fichier server.xml : ceci permet de les distinguer car par défaut, il possède le même nom (Catalina).

L'attribut reloadable doit être à false pour chaque context : ceci évite à Tomcat de vérifier périodiquement le besoin de recharger les classes.

Il faut désactiver dans le fichier server.xml les connectors qui ne sont pas utilisés.

71.12. La sécurisation du serveur

Cette section présente rapidement quelques actions possibles pour améliorer la sécurisation d'un serveur Tomcat notamment dans une optique d'exécution dans un environnement de production. Ces actions ne concernent que Tomcat et occulte complètement la sécurisation du système et du réseau.

Il faut exécuter Tomcat avec un user qui dispose uniquement des privilèges requis pour l'exécution (par exemple, il ne faut surtout pas exécuter Tomcat avec le user root sous Unix mais créer un user tomcat dédié à son exécution).

Les user possédant un rôle admin doivent avoir un mot de passe non triviale : il faut prohiber les user/mot de passe de type admin/admin.

Les droits d'accès aux répertoires et fichiers de Tomcat doivent être vérifiés pour ne pas permettre à quiconque de les modifier.

Il est préférable de ne pas installer l'outil d'administration : les fichiers de configuration doivent être modifiés à la main via le système.

Il faut modifier les valeurs par défaut des attributs port et shutdown du tag <server> du fichier de configuration server.xml : ceci permet d'éviter un éventuel shutdown grâce aux valeurs par défaut.

72. Des outils open source pour faciliter le développement

La communauté open source propose de nombreuses bibliothèques mais aussi des outils dont le but est de faciliter le travail des développeurs. Certains de ces outils sont détaillés dans des chapitres dédiés notamment Ant, Maven et JUnit. Ce chapitre va présenter d'autres outils open source pouvant être regroupés dans plusieurs catégories : contrôle de la qualité des sources et génération et mis en forme de code.

La génération de certains morceaux de code ou de fichiers de configuration peut parfois être fastidieuse voir même répétitif dans certains cas. Pour faciliter le travail des développeurs, des outils open source ont été développés par la communauté afin de générer certains morceaux de code. Ce chapitre présente deux outils open source : XDoclet et Middlegen.

La qualité du code source est un facteur important pour tous développements. Ainsi certains outils permettent de faire des vérifications sur des règles de codification dans le code source. C'est le cas pour l'outil CheckStyle.

Ce chapitre contient plusieurs sections :

- CheckStyle
- Jalopy
- XDoclet
- Middlegen

72.1. CheckStyle

CheckStyle est un outil open source qui propose de puissantes fonctionnalités pour appliquer des contrôles sur le respect de règles de codifications.

Pour définir les contrôles réalisés lors de son exécution CheckStyle utilise une configuration qui repose sur des modules. Cette configuration est définie dans un fichier XML qui précise les modules utilisés et pour chacun d'entre eux leurs paramètres.

Le plus simple est d'utiliser le fichier de configuration nommé sun_checks.xml fourni avec CheckStyle. Cette configuration propose d'effectuer des contrôles de respect des normes de codification proposée par Sun. Il est aussi possible de définir son propre fichier de configuration.

Le site officiel de CheckStyle est à l'URL : http://checkstyle.sourceforge.net/

La version utilisée dans cette section est la 3.4

72.1.1. L'installation

Il faut télécharger le fichier checkstyle-3.4.zip sur le site de ChekStyle et le décompresser dans un répertoire du système.

La décompression de ce fichier créé un répertoire checkstyle-3.4 contenant lui même les bibliothèques utiles et deux répertoires (docs et contrib).

CheckStyle peut s'utiliser de deux façons :

- en ligne de commande
- comme une tâche Ant ce qui permet d'automatiser son exécution

72.1.2. L'utilisation avec Ant

Pour utiliser CheckStyle, le plus simple est d'ajouter la bibliothèque checkstyle-all-3.4.jar au classpath.

Dans les exemples de cette section, la structure de répertoires suivante est utilisée :

/bin
/lib
/outils
/outils/lib
/outils/checkstyle
/src
/temp
/temp/checkstyle

Le fichier checkstyle-all-3.4.jar est copié dans le répertoire outils/lib et le fichier sun_checks.xml fourni par CheckStyle est copié dans le répertoire outils/checkstyle.

Il faut déclarer le tag CheckStyle dans Ant en utilisant le tag <taskdef> et définir une tâche qui va utiliser le tag <CheckStyle>.

Exemple : fichier source de test

```
public class MaClasse {

  public static void main() {
    System.out.println("Bonjour");
  }

}
```

Le fichier de build ci-dessous sera exécuté par Ant.

Exemple :

```
<project name="utilisation de checkstyle" default="compile" basedir=".">
  <!-- Definition des proprietes du projet -->
  <property name="projet.sources.dir"       value="src"/>
  <property name="projet.bin.dir"           value="bin"/>
  <property name="projet.lib.dir"           value="lib"/>
  <property name="projet.temp.dir"          value="temp"/>
  <property name="projet.outils.dir"        value="outils"/>
  <property name="projet.outils.lib.dir"    value="${projet.outils.dir}/lib"/>

  <!-- Definition du classpath du projet -->
  <path id="projet.classpath">
    <fileset dir="${projet.lib.dir}">
      <include name="*.jar"/>
    </fileset>
    <fileset dir="${projet.outils.lib.dir}">
      <include name="*.jar"/>
    </fileset>
    <pathelement location="${projet.bin.dir}" />
  </path>

  <!-- Declaration de la tache Ant permettant l'execution de checkstyle -->
  <taskdef resource="checkstyletask.properties"
    classpathref="projet.classpath" />
```

```
  <!-- execution de checkstyle -->
  <target name="checkstyle" description="CheckStyle">
    <checkstyle config="outils/checkstyle/sun_checks.xml">
      <fileset dir="${projet.sources.dir}" includes="**/*.java"/>
      <formatter type="plain"/>
    </checkstyle>
  </target>

  <!-- Compilation des classes du projet -->
  <target name="compile" depends="checkstyle" description="Compilation des classes">
    <javac srcdir="${projet.sources.dir}"
      destdir="${projet.bin.dir}"
      debug="on"
      optimize="off"
      deprecation="on">
      <classpath refid="projet.classpath"/>
    </javac>
  </target>

</project>
```

Résultat :

```
C:\java\test\testcheckstyle>ant
Buildfile: build.xml
checkstyle:
[checkstyle] C:\java\test\testcheckstyle\src\package.html:0: Missing package doc
umentation file.
[checkstyle] C:\java\test\testcheckstyle\src\MaClasse.java:0: File does not end
with a newline.
[checkstyle] C:\java\test\testcheckstyle\src\MaClasse.java:2: Missing a Javadoc
comment.
[checkstyle] C:\java\test\testcheckstyle\src\MaClasse.java:2:1: Utility classes
should not have a public or default constructor.
[checkstyle] C:\java\test\testcheckstyle\src\MaClasse.java:4:3: Missing a Javado
c comment.
[checkstyle] C:\java\test\testcheckstyle\src\MaClasse.java:5: Line has trailing
spaces.
[checkstyle] C:\java\test\testcheckstyle\src\MaClasse.java:5:35: La ligne contie
nt un caractÞre tabulation.
[checkstyle] C:\java\test\testcheckstyle\src\MaClasse.java:7: Line has trailing
spaces.
[checkstyle] C:\java\test\testcheckstyle\src\MaClasse.java:7:1: La ligne contien
t un caractÞre tabulation.
BUILD FAILED
file:C:/java/test/testcheckstyle/build.xml:27: Got 9 errors.
      Total time: 7 seconds
```

Le tag <checkstyle> possède plusieurs attributs :

Nom	Rôle
file	précise le nom de l'unique fichier à vérifier. Pour préciser un ensemble de fichiers, il faut définir un ensemble de fichiers grâce à un tag fils <fileset>
config	précise le nom du fichier de configuration des modules de ChecksStyle
failOnViolation	précise si les traitements de l'outil Ant doivent être stoppés en cas d'échec des contrôles. La valeur par défaut est true
failureProperty	précise le nom d'une propriété qui sera valorisée en cas d'échec des contrôles

Exemple :

```
...
  <!-- execution de checkstyle -->
  <target name="checkstyle" description="CheckStyle">
    <checkstyle config="outils/checkstyle/sun_checks.xml" failOnViolation="false">
      <fileset dir="${projet.sources.dir}" includes="**/*.java"/>
```

```
      <formatter type="plain"/>
    </checkstyle>
  </target>
...
```

Résultat :

```
C:\java\test\testcheckstyle>ant
Buildfile: build.xml
checkstyle:
[checkstyle] C:\java\test\testcheckstyle\src\package.html:0: Missing package doc
umentation file.
[checkstyle] C:\java\test\testcheckstyle\src\MaClasse.java:2: Missing a Javadoc
comment.
[checkstyle] C:\java\test\testcheckstyle\src\MaClasse.java:2:1: Utility classes
should not have a public or default constructor.
[checkstyle] C:\java\test\testcheckstyle\src\MaClasse.java:4:3: Missing a Javado
c comment.
compile:
[javac] Compiling 1 source file to C:\java\test\testcheckstyle\bin
BUILD SUCCESSFUL
      Total time: 11 seconds
```

Il est possible de préciser deux types de format de sortie des résultats lors de l'exécution de CheckStyle. Le format de sortie des résultats est précisé par un tag fils <formatter>. Ce tag possède deux attributs :

Nom	Rôle
type	précise le type. Deux valeurs sont possibles : plain (par défaut) et xml
toFile	précise un fichier qui va contenir les résultats dont le format correspondra au type (par défaut la sortie standard de la console)

Exemple :

```
...
<!-- execution de checkstyle -->
<target name="checkstyle" description="CheckStyle">
  <checkstyle config="outils/checkstyle/sun_checks.xml" failOnViolation="false">
    <fileset dir="${projet.sources.dir}" includes="**/*.java"/>
    <formatter type="xml" toFile="${projet.temp.dir}/checkstyle_erreurs.xml"/>
  </checkstyle>
</target>
...
```

Il est alors possible d'appliquer une feuille de style sur le fichier XML généré afin de créer un rapport dans un format dédié. L'exemple suivant utilise une feuille de style fournie par CheckStyle dans le répertoire contrib : cette feuille, nommée checkstyle-frames.xsl, est copiée dans le répertoire outils/checkstyle.

Exemple :

```
...
<!-- execution de checkstyle -->
<target name="checkstyle" description="CheckStyle">
  <checkstyle config="${projet.outils.dir}/checkstyle/sun_checks.xml" failOnViolation="false">
    <fileset dir="${projet.sources.dir}" includes="**/*.java"/>
    <formatter type="xml" toFile="${projet.temp.dir}/checkstyle/checkstyle_erreurs.xml"/>
  </checkstyle>
  <style in="${projet.temp.dir}/checkstyle/checkstyle_erreurs.xml"
    out="${projet.temp.dir}/checkstyle/checkstyle_rapport.htm"
    style="${projet.outils.dir}/checkstyle/checkstyle-frames.xsl"/>
</target>
...
```

Exemple :

```
C:\java\test\testcheckstyle>ant
Buildfile: build.xml
checkstyle:
[style] Processing C:\java\test\testcheckstyle\temp\checkstyle\checkstyle_er
reurs.xml to C:\java\test\testcheckstyle\temp\checkstyle\checkstyle_rapport.htm
[style] Loading stylesheet C:\java\test\testcheckstyle\outils\checkstyle\che
ckstyle-frames.xsl
compile:
BUILD SUCCESSFUL
Total time: 8 seconds
```

Il est possible d'utiliser d'autres feuilles de style fournies par CheckStyle ou de définir sa propre feuille de style.

72.1.3. L'utilisation en ligne de commandes

Pour utiliser CheckStyle en ligne de commandes, il faut ajouter le fichier checkstyle-all-3.4.jar au classpath par exemple en utilisant l'option -cp de l'interpréteur Java.

La classe à exécuter est com.puppycrawl.tools.checkstyle.Main

CheckStyle accepte plusieurs paramètres pour son exécution :

Option	Rôle
-c fichier_de_configuration	précise le fichier de configuration
-f format	précise le format (plain ou xml)
-o fichier	précise le fichier qui va contenir les résultats
-r	précise le répertoire dont les fichiers sources vont être récursivement traités

Exemple :

```
java -cp outils\lib\checkstyle-all-3.4.jar com.puppycrawl.tools.checkstyle.Main
  -c outils\checkstyle/sun_checks.xml -r src
```

Exemple :

```
...
C:\java\test\testcheckstyle>java -cp outils\lib\checkstyle-all-3.4.jar com.puppy
crawl.tools.checkstyle.Main -c outils\checkstyle/sun_checks.xml -r src
Starting audit...
C:\java\test\testcheckstyle\src\package.html:0: Missing package documentation fi
le.
src\MaClasse.java.bak:0: File does not end with a newline.
src\MaClasse.java:2: Missing a Javadoc comment.
src\MaClasse.java:2:1: Utility classes should not have a public or default const
ructor.
src\MaClasse.java:4:3: Missing a Javadoc comment.
Audit done.
...
```

72.2. Jalopy

Jalopy est un outil open source qui propose de formater les fichiers source selon des règles définies.

Le site web officiel de Jalopy est http://jalopy.sourceforge.net

Jalopy propose entre autre les fonctionnalités suivantes :

- formatage des accolades selon plusieurs formats (C, Sun, GNU)
- indentation du code
- gestion des sauts de ligne
- génération automatique ou vérification des commentaires Javadoc
- ordonnancement des éléments qui composent la classe
- ajout de texte au début et la fin de chaque fichier
- des plug-ins pour une intégration dans plusieurs outils : ant, Eclipse, Jbuilder, ...
- ...

Pour connaître les règles de formatage à appliquer, Jalopy utilise une convention qui est un ensemble de paramètres.

Jalopy peut être utilisé grâce à ces plug-ins de plusieurs façons notamment avec Ant, en ligne commande ou avec certains IDE.

La version de Jalopy utilisée dans cette section est la 1.0.B10

72.2.1. L'utilisation avec Ant

Le plus simple est d'utiliser Jalopy avec Ant pour automatiser son utilisation : pour cela, il faut télécharger le fichier jalopy-ant-0.6.2.zip et le décompresser dans un répertoire du système.

La structure de l'arborescence du projet utilisé dans cette section est la suivante :

/bin
/lib
/outils
/outils/lib
/src

Le répertoire src contient les sources Java à formater.

Les fichiers du répertoire lib de Jalopy sont copiés dans le répertoire outils/lib du projet.

Exemple : le fichier source qui sera formaté

```
public class MaClasse {public static void main() { System.out.println("Bonjour"); }}
```

Il faut définir un fichier build.xml pour Ant qui va contenir les différentes tâches du projet dont une permettant l'appel à Jalopy.

Exemple :

```
<project name="utilisation de jalopy" default="jalopy" basedir=".">
  <!-- Definition des proprietes du projet -->
  <property name="projet.sources.dir" value="src"/>
  <property name="projet.bin.dir" value="bin"/>
  <property name="projet.lib.dir" value="lib"/>
  <property name="projet.temp.dir" value="temp"/>
  <property name="projet.outils.dir" value="outils"/>
  <property name="projet.outils.lib.dir" value="${projet.outils.dir}/lib"/>

  <!-- Definition du classpath du projet -->
  <path id="projet.classpath">
    <fileset dir="${projet.lib.dir}">
      <include name="*.jar"/>
    </fileset>
    <fileset dir="${projet.outils.lib.dir}">
```

```
      <include name="*.jar"/>
    </fileset>
    <pathelement location="${projet.bin.dir}" />
  </path>

  <!-- Declaration de la tache Ant permettant l'execution de jalopy -->
  <taskdef name="jalopy"
         classname="de.hunsicker.jalopy.plugin.ant.AntPlugin"
      classpathref="projet.classpath" />

  <!-- execution de jalopy -->
  <target name="jalopy" description="Jalopy" depends="compile" >
    <jalopy loglevel="info"
                threads="2"
           classpathref="projet.classpath">
      <fileset dir="${projet.sources.dir}">
        <include name="**/*.java" />
      </fileset>
    </jalopy>
  </target>

  <!-- Compilation des classes du projet -->
  <target name="compile" description="Compilation des classes">
    <javac srcdir="${projet.sources.dir}"
             destdir="${projet.bin.dir}"
               debug="on"
            optimize="off"
         deprecation="on">
      <classpath refid="projet.classpath"/>
    </javac>
  </target>

</project>
```

Exemple :

```
C:\java\test\testjalopy>ant
Buildfile: build.xml

compile:
    [javac] Compiling 1 source file to C:\java\test\testjalopy\bin

jalopy:
   [jalopy] Jalopy Java Source Code Formatter 1.0b10
   [jalopy] Format 1 source file
   [jalopy] C:\java\test\testjalopy\src\MaClasse.java:0:0: Parse
   [jalopy] 1 source file formatted

BUILD SUCCESSFUL
Total time: 9 seconds
```

Suite à l'exécution de Jalopy, le code du fichier est reformaté.

Exemple :

```
public class MaClasse {
    public static void main() {
        System.out.println("Bonjour");
    }
}
```

Il est fortement recommandé de réaliser la tâche de compilation des sources avant leur formatage car pour assurer un formatage correct les sources doivent être correctes syntaxiquement parlant.

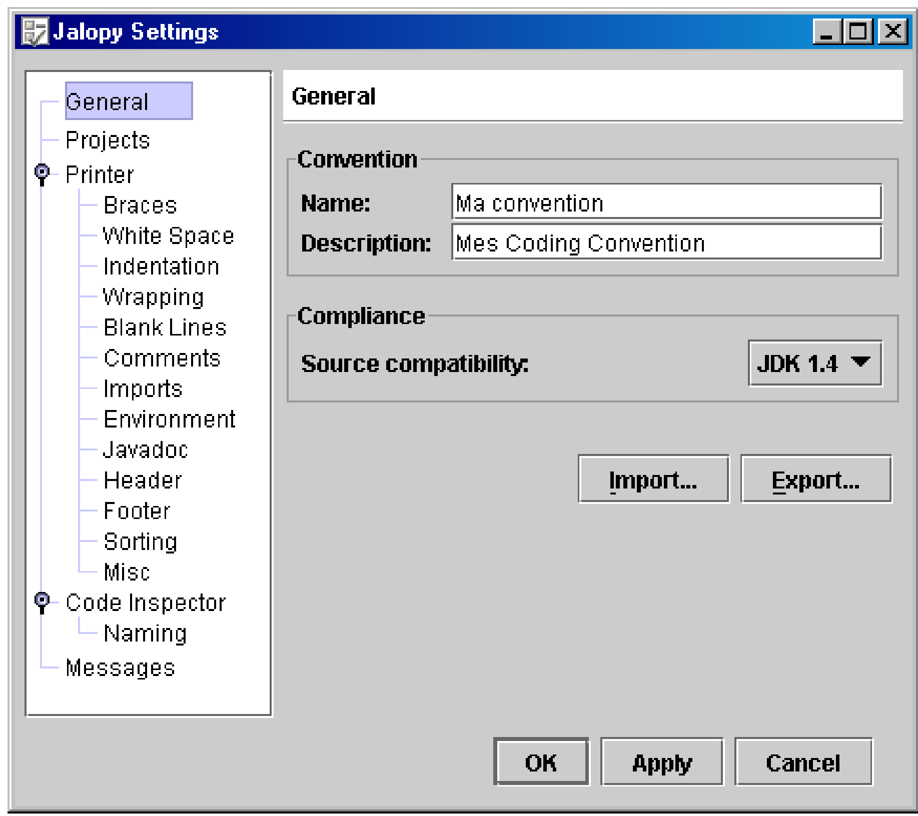
72.2.2. Les conventions

Jalopy est hautement paramétrable. Les options utilisées sont regroupées dans une convention.

Tous ses paramètres sont stockés dans le sous répertoire .jalopy du répertoire Home de l'utilisateur.

Pour faciliter la gestion de ces paramètres, Jalopy propose un outil graphique qui permet de gérer les conventions.

Pour exécuter cet outil, il suffit de lancer le script preferences dans le répertoire bin de Jalopy (preferences.bat sous Windows et preferences.sh sous Unix).



Toutes les nombreuses options de formatage d'une convention peuvent être réglées via cet outil. Consultez la documentation fournie avec Jalopy pour un détail de chaque option.

Une fonctionnalité particulièrement utile de cet outil est de proposer une prévisualisation d'un exemple mettant en oeuvre les options sélectionnées.

Voici un exemple avec quelques personnalisations notamment une gestion des clauses import, la génération des commentaires Javadoc :

Exemple :

```
import java.util.*;

public class MaClasse {
    public static void main() {
        List liste = new ArrayList();
        System.out.println("Bonjour");
    }

    /**
     *
     */
    private int maMethode(int valeur) {
        return valeur * 2;
    }
}
```

Résultat :

```
C:\java\test\testjalopy>ant
Buildfile: build.xml

compile:
    [javac] Compiling 1 source file to C:\java\test\testjalopy\bin

jalopy:
   [jalopy] Jalopy Java Source Code Formatter 1.0b10
   [jalopy] Format 1 source file
   [jalopy] C:\java\test\testjalopy\src\MaClasse.java:0:0: Parse
   [jalopy] C:\java\test\testjalopy\src\MaClasse.java:1:0: On-demand import "jav
a.util.List" expanded
   [jalopy] C:\java\test\testjalopy\src\MaClasse.java:1:0: On-demand import "jav
a.util.ArrayList" expanded
   [jalopy] C:\java\test\testjalopy\src\MaClasse.java:11:1: Generated Javadoc co
mment
   [jalopy] C:\java\test\testjalopy\src\MaClasse.java:18:3: Generated Javadoc co
mment
   [jalopy] 1 source file formatted

BUILD SUCCESSFUL
Total time: 10 seconds
```

Voici le source du code reformaté :

Exemple :

```
//==============================================================================
// fichier :       MaClasse.java
// projet :        $project$
//
// Modification :  date :       $Date$
//                 auteur :     $Author$
//                 revision :   $Revision$
//------------------------------------------------------------------------------
// copyright:   JMD
//==============================================================================

import java.util.ArrayList;
import java.util.List;


/**
 * DOCUMENT ME!
 *
 * @author $author$
 * @version $Revision$
 */
public class MaClasse {
  /**
   * DOCUMENT ME!
   */
  public static void main() {
    List liste = new ArrayList();
    System.out.println("Bonjour");
    System.out.println("");
  }

  /**
   * Calculer le double
   *
   * @param valeur DOCUMENT ME!
   *
   * @return DOCUMENT ME!
   */
  private int maMethode(int valeur) {
    return valeur * 2;
  }
}
```

```
// fin du fichier
```

72.3. XDoclet

La suite de ce chapitre sera développée dans une version future de ce document

72.4. Middlegen

La suite de ce chapitre sera développée dans une version future de ce document

Pour faciliter le développement d'applications, il est préférable d'utiliser une méthodologie pour l'analyse et d'utiliser ou de définir des normes lors du développement. L'utilisation de frameworks et de bibliothèques dans une architecture adaptée fait aussi parti des impératifs à mettre en oeuvre lors du développement d'une application.

La communauté Java permet d'obtenir des outils, des frameworks, des bibliothèques mais aussi de très nombreuses informations autour des technologies Java.

Cette partie contient les chapitres suivants :

- Java et UML : propose une présentation de la notation UML ainsi que sa mise en oeuvre avec Java
- Les motifs de conception (design patterns) : présente certains modèles de conception en programmation orienté objet et leur mise en oeuvre avec Java
- Des normes de développement : propose de sensibiliser le lecteur à l'importance de la mise en place de normes de développement sur un projet et propose quelques règles pour définir une telle norme
- L'encodage des caractères : ce chapitre fournit des informations sur l'encodage des caractères dans les applications Java.
- Les frameworks : présente les frameworks et propose quelques solutions open source pour quelques unes de leur famille
- Les frameworks de tests : propose une présentation de frameworks et outils pour faciliter les tests du code
- JUnit : présente en détail le framework de tests unitaires le plus utilisé

- Les objets de type Mock : ce chapitre détaille la mise en oeuvre des objets de type mocks et les doublures d'objets
- Des bibliothèques open source : présentation de quelques bibliothèques de la communauté open source particulièrement pratiques et utiles
- La génération de documents : Ce chapitre présente plusieurs API open source permettant la génération de documents dans différents formats notamment PDF et Excel
- La validation des données : La validation des données est une tâche commune,nécessaire et importante dans chaque application de gestion de données.
- La communauté Java : ce chapitre présente quelques unes des composantes de l'imposante communauté Java

73. Java et UML

Chapitre 7 3

La suite de ce chapitre sera développée dans une version future de ce document

Le but d'UML est de modéliser un système en utilisant des objets. L'orientation objet de Java ne peut qu'inciter à l'utiliser avec UML. La modélisation proposée par UML repose sur 9 diagrammes.

Ce chapitre contient plusieurs sections :

- La présentation d'UML
- Les commentaires
- Les cas d'utilisation (uses cases)
- Le diagramme de séquence
- Le diagramme de collaboration
- Le diagramme d'états-transitions
- Le diagramme d'activités
- Le diagramme de classes
- Le diagramme d'objets
- Le diagramme de composants
- Le diagramme de déploiement

73.1. La présentation d'UML

UML qui est l'acronyme d'Unified Modeling Language est aujourd'hui indissociable de la conception objet. UML est le résultat de la fusion de plusieurs méthodes de conception objet des pères d'UML étaient les auteurs : Jim Rumbaugh (OMT), Grady Booch (Booch method) et Ivar Jacobson (use case).

UML a adopté et normalisé par l'OMG (Object Management Group) en 1997.

D'une façon général, UML est une représentation standardisée d'un système orienté objet.

UML n'est pas une méthode de conception mais notation graphique normalisée de présentation de certains concepts pour modéliser des systèmes objets. En particulier, UML ne précise pas dans quel ordre et comment concevoir les différents diagrammes qu'il défini. Cependant, UML est indépendant de toute méthode de conception et peut être utilisé avec n'importe lequel de ces processus.

Un standard de présentation des concepts permet de faciliter le dialogue entre les différents autres acteurs du projet : les autres analystes, les développeurs, et même les utilisateurs.

UML est composé de neuf diagrammes :

- des cas d'utilisation
- de séquence
- de collaboration
- d'états-transitions
- d'activité
- de classes
- d'objets
- de composants
- de déploiement

UML regroupe ces neufs diagrammes dans trois familles :

- les diagrammes statiques (diagrammes de classes, d'objet et de cas d'utilisation)
- les diagrammes dynamiques (diagrammes d'activité, de collaboration, de séquence, d'état-transitions et de cas d'utilisation)
- les diagrammes d'architecture : (diagrammes de composants et de déploiements)

73.2. Les commentaires

Utilisable dans chaque diagramme, UML propose une notation particulière pour indiquer des commentaires.

Exemple :

73.3. Les cas d'utilisation (uses cases)

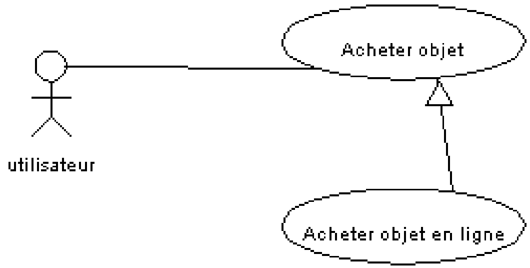
Ils sont développés par Ivar Jacobson et permettent de modéliser des processus métiers en les découpant en cas d'utilisation.

Ce diagramme permet de représenter les fonctionnalités d'un système. Il se compose :

- d'acteurs : ce sont des entités qui utilisent le système à représenter
- les cas d'utilisation : ce sont des fonctionnalités proposées par le système

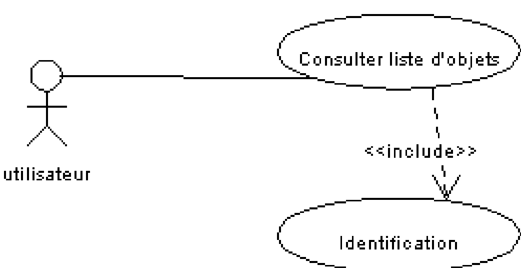
Un acteur n'est pas une personne désignée : c'est une entité qui joue un rôle dans le système. Il existe plusieurs types de relations qui associent un acteur et un cas d'utilisation :

- la généralisation : cette relation peut être vue comme une relation d'héritage. Un cas d'utilisation enrichie un autre cas en le spécialisant



- l'extension (stéréotype <<extend>>) : le cas d'utilisation complète un autre cas d'utilisation

* l'inclusion (stéréotype <<include>>) : le cas d'utilisation utilise un autre cas d'utilisation



Les cas d'utilisation sont particulièrement intéressants pour recenser les différents acteurs et les différentes fonctionnalités d'un système.

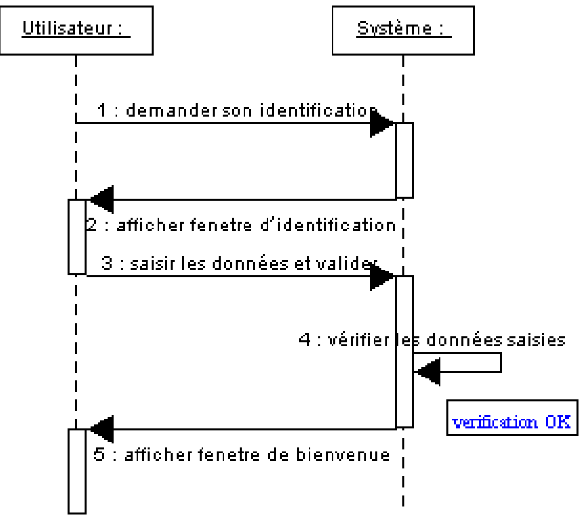
La simplicité de ce diagramme lui permet d'être rapidement compris par des utilisateurs non informaticiens. Il est d'ailleurs très important de faire participer les utilisateurs tout au long de son évolution.

Le cas d'utilisation est ensuite détaillé en un ou plusieurs scénarios. Un scénario est une suite d'échanges entre des acteurs et le système pour décrire un cas d'utilisation dans un contexte particulier. C'est est un enchainement précis et ordonné d'opérations pour réaliser le cas d'utilisation.

Si le scénario est trop "volumineux", il peut être judicieux de découper le cas d'utilisation en plusieurs cas d'utilisation et d'utiliser les relations appropriées.

Un scénario peut être représenté par un diagramme de séquence ou sous une forme textuelle. La première forme est très visuelle

Exemple :



La seconde facilite la représentation des opérations alternatives.

Les cas d'utilisation permettent de modéliser des concepts fonctionnels. Il ne précise pas comment chaque opération sera implémentée techniquement. Il faut rester le plus abstrait possible dans les concepts qui s'approchent de la partie technique.

Le découpage d'un système en cas d'utilisation n'est pas facile car il faut trouver un juste milieu entre un découpage faible (les scénarios sont importants) et un découpage faible (les cas d'utilisation se réduisent à une seule opération).

73.4. Le diagramme de séquence

Il permet de modéliser les échanges de messages entre les différents objets dans le contexte d'un scénario précis

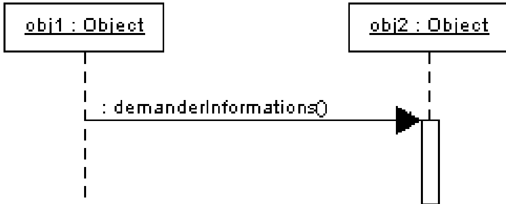
Il permet de représenter les interactions entre différentes entités. Il s'utilise essentiellement pour décrire les scénarios d'un cas d'utilisation (les entités sont les acteurs et le système) ou décrire des échanges entre objets.

Dans le premier cas, les interactions sont des actions qui sont réalisées par une entité.

Dans le second cas, les itérations sont des appels de méthode.

Les itérations peuvent être de deux types :

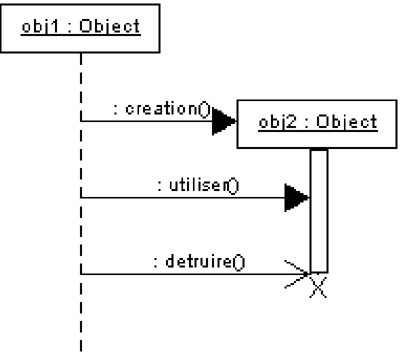
- synchrone : l'émetteur attend une réponse du récepteur



- asynchrone : l'émetteur poursuit son exécution sans attendre de réponse

Un diagramme de séquence peut aussi représenter le cycle de vie d'un objet.

Exemple :



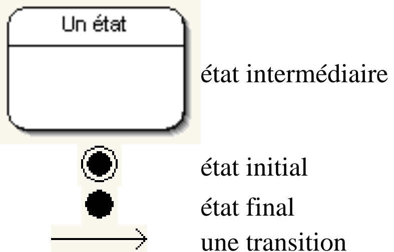
73.5. Le diagramme de collaboration

Il permet de modéliser la collaboration entre les différents objets.

73.6. Le diagramme d'états-transitions

Un diagramme d'état permet de modéliser les différents états d'une entité, en générale une classe. L'ensemble de ces états est connu.

Ce diagramme se compose de plusieurs éléments principaux :



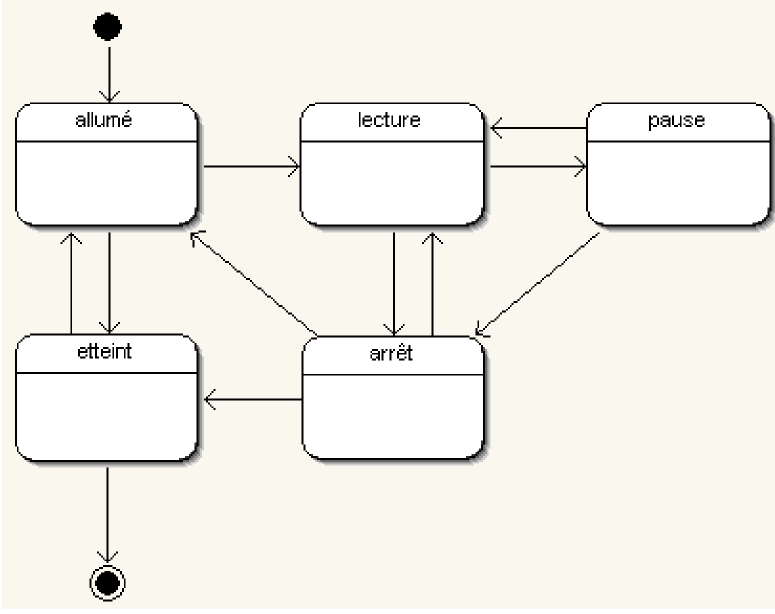
état intermédiaire

état initial

état final

une transition

Exemple :

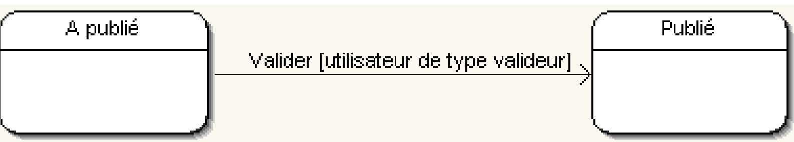


Les transitions sont des événements qui permettent de passer d'un état à un autre : chaque transition possède un sens qui précise l'état de départ et l'état d'arrivée (du côté de la flèche). Une transition peut avoir un nom qui permet de la préciser.

Exemple :

Il est possible d'ajouter une condition à une transition qui est expression booléenne qui sera vérifiée lors d'une demande de transition. Cette condition est indiquée entre crochet.

Exemple :



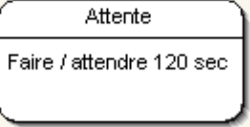
Dans un état, il est possible de préciser des actions ou des activités qui sont des traitements à réaliser dans un état. Celles ci sont décrites avec une étiquette qui désigne le moment de l'exécution.

Une action est un traitement cours. Une activité est un traitement durant tout ou partie de la durée de maintient de l'état.

Plusieurs étiquettes standards sont définies :

- entrée (entry) : action réalisé à l'entrée dans l'état
- sortie (exit) : action réalisée à la sortie de l'état
- faire (do) : activité exécuté durant l'état

Exemple :



Il est aussi possible de définir des actions internes

73.7. Le diagramme d'activités

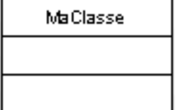
73.8. Le diagramme de classes

Ce schéma représente les différentes classes : il détaille le contenu de chaque classe mais aussi les relations qui peuvent exister entre les différentes classes.

Une classe est représentée par un rectangle séparée en trois parties :

- la première partie contient le nom de la classe
- la seconde contient les attributs de la classe
- la dernière contient les méthodes de la classe

Exemple :



Exemple :

```
public class MaClasse {
}
```

Les attributs d'une classe

Pour définir un attribut, il faut préciser son nom suivi du caractère ":" et du type de l'attribut.

Le modificateur d'accès de l'attribut doit précéder son nom et peut prendre les valeurs suivantes :

Caractère	Rôle
+	accès public
#	accès protected
-	accès private

Une valeur d'initialisation peut être précisée juste après le type en utilisant le signe "=" suivi de la valeur.

Exemple :

MaClasse
+champPublic : int = 0
#champProtected : double = 0
-champPrive : boolean = false

Exemple :

```
public class MaClasse {

  public int champPublic = 0;
  protected double champProtected = 0;
  private boolean champPrive = false;
}
```

Les méthodes d'une classe

Les modificateurs sont identiques à ceux des attributs.

Les paramètres de la méthode peuvent être précisés en les indiquant entre les parenthèses sous la forme nom : type.

Si la méthode renvoie une valeur son type doit être précisé après un signe ":".

Exemple :

MaClasse
+methode1(valeur:int)
#methode2() : String
-methode3()

Exemple :

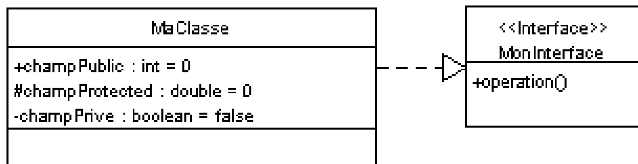
```
public class MaClasse {

  public void methode1(int valeur){
  }

  protected String methode2(){
  }

  private void methode3(){
```

```
    }
}
```

Il n'est pas obligatoire d'inclure dans le diagramme tous les attributs et toutes les méthodes d'une classe : seules les entités les plus significatives et utiles peuvent être mentionnées.

L'implémentation d'une interface

Exemple :

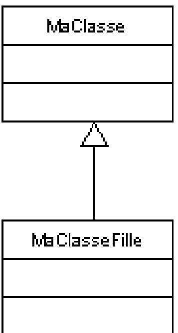


Exemple :

```
public class MaClasse implements MonInterface {
    public int champPublic = 0;
    protected double champProtected = 0;
    private boolean champPrive = false;

    public operation() {

    }
}
```

La relation d'héritage

Exemple :



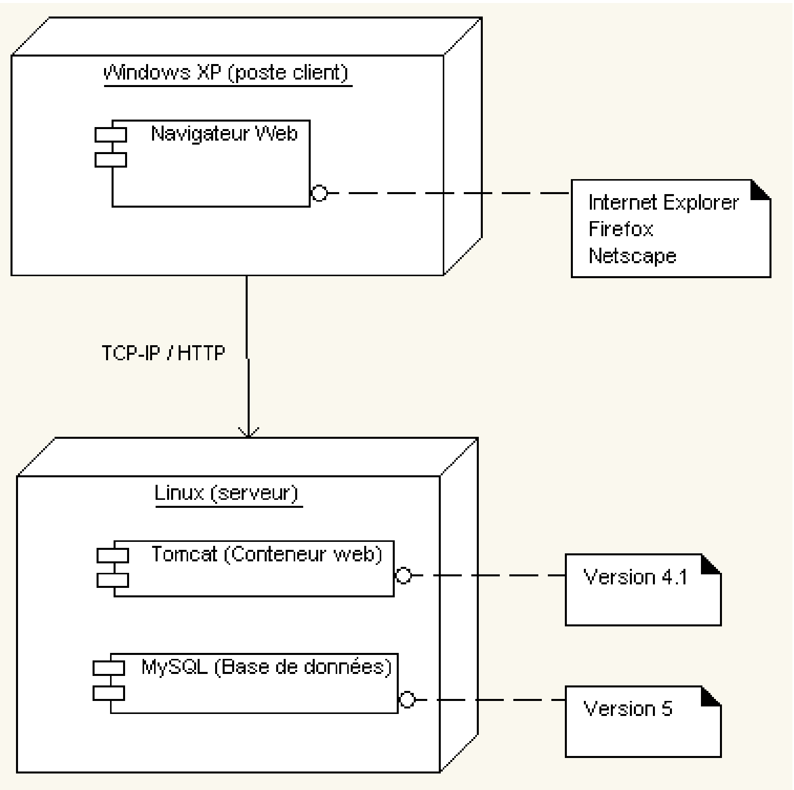
Exemple :

```
public class MaClasseFille extends MaClasse {

}
```

73.9. Le diagramme d'objets

73.10. Le diagramme de composants

73.11. Le diagramme de déploiement



74. Les motifs de conception (design patterns)

Le nombre de développement avec des technologies orientées objets augmentant, l'idée de réutiliser des techniques pour solutionner des problèmes courants à abouti aux recensements d'un certain nombre de modèles connus sous les motifs de conception (design patterns).

Ces modèles sont définis pour pouvoir être utilisés avec un maximum de langage orienté objet.

Le nombre de ces modèles est en constante augmentation. Le but de ce chapitre n'est pas de tous les recenser mais de présenter les plus utilisés et de fournir un ou des exemples de leur mise en oeuvre avec Java.

Il est habituel de regrouper ces modèles communs dans trois grandes catégories :

- les modèles de création (creational patterns)
- les modèles de structuration (structural patterns)
- les modèles de comportement (behavioral patterns)

Le motif de conception le plus connu est surement le modèle MVC (Model View Controller) mis en oeuvre en premier avec SmallTalk.

Ce chapitre contient plusieurs sections :

- ♦ Les modèles de création
- ♦ Les modèles de structuration
- ♦ Les modèles de comportement

74.1. Les modèles de création

Dans cette catégorie, il existe 5 modèles principaux :

Nom	Rôle
Fabrique (Factory)	Créer un objet dont le type dépend du contexte
Fabrique abstraite (abstract Factory)	Fournir une interface unique pour instancier des objets d'une même famille sans avoir à connaître les classes à instancier
Monteur (Builder)	
Prototype (Prototype)	Création d'objet à partir d'un prototype
Singleton (Singleton)	Classe qui ne pourra avoir qu'une seule instance

74.1.1. Fabrique (Factory)

La fabrique permet de créer un objet dont le type dépend du contexte : cet objet fait partie d'un ensemble de sous classes. L'objet retourné par la fabrique est donc toujours du type de la classe mère mais grâce au polymorphisme les traitements exécutés sont ceux de l'instance créée.

Ce motif de conception est utilisé lorsqu'à l'exécution il est nécessaire de déterminer dynamiquement quel objet d'un ensemble de sous classes doit être instancié.

Il est utilisable lorsque :

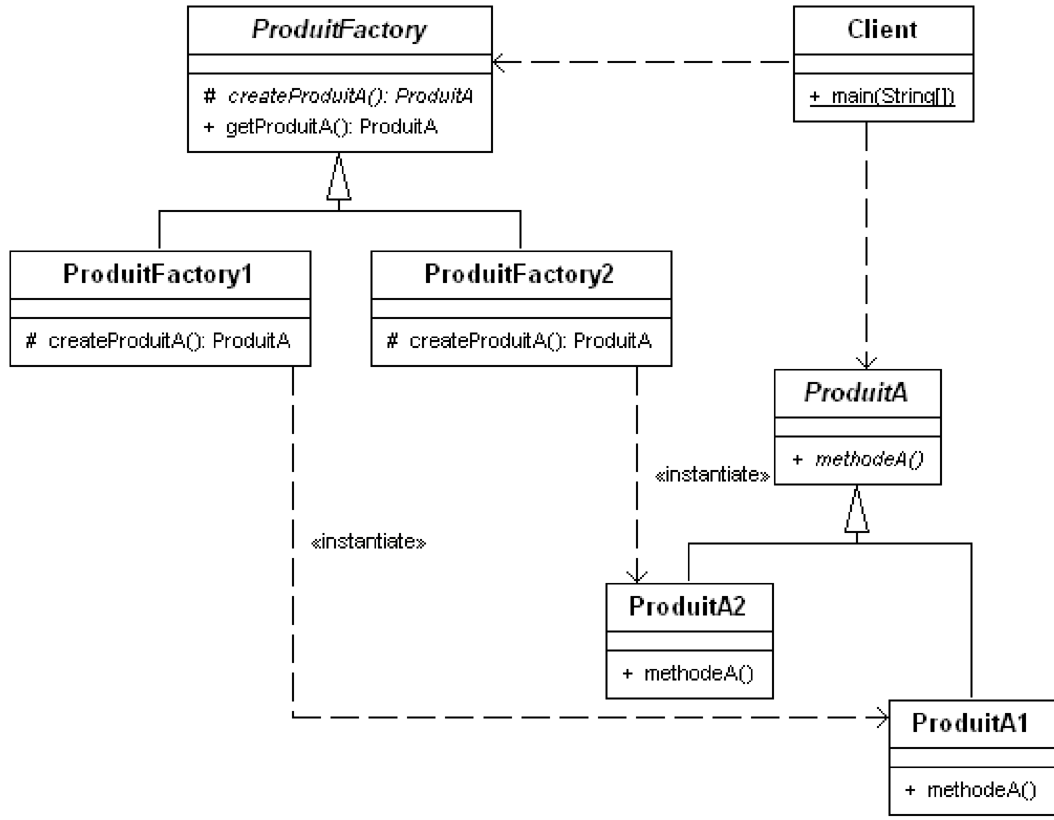
- Le client ne peut déterminer le type d'objet à créer qu'à l'exécution
- Il y a une volonté de centraliser la création des objets

L'utilisation d'une fabrique permet de rendre l'instanciation d'objets plus flexible que l'utilisation de l'opérateur d'instanciation new.

Ce design pattern peut être implémenté sous plusieurs formes dont les deux principales sont :

- Déclarer la fabrique abstraite et laisser une de ces sous classes créer l'objet
- Déclarer une fabrique dont la méthode de création de l'objet attend les données nécessaires pour déterminer le type de l'objet à instancier

Il est possible d'implémenter la fabrique sous la forme d'une classe abstraite et de définir des sous classes chargées de réaliser les différentes instanciations.



La classe ProduitFactory propose la méthode getProduitA() qui se charge de retourner l'instance créée. Elle se charge uniquement de retourner la valeur de la méthode createProduitA().

Les classes ProduitFactory1 et ProduitFactory2 sont les implémentations concrètes de la fabrique. Elles redéfinissent la méthode createProduitA() pour qu'elle renvoie l'instance du produit.

La classe ProduitA est la classe abstraite mère de tous les produits.

Les classes ProduitA1 et ProduitA2 sont des implémentations concrètes de produit.

Exemple : le code sources des différentes classes

```
package com.jmd.test.dej.factory1;

public class Client {
```

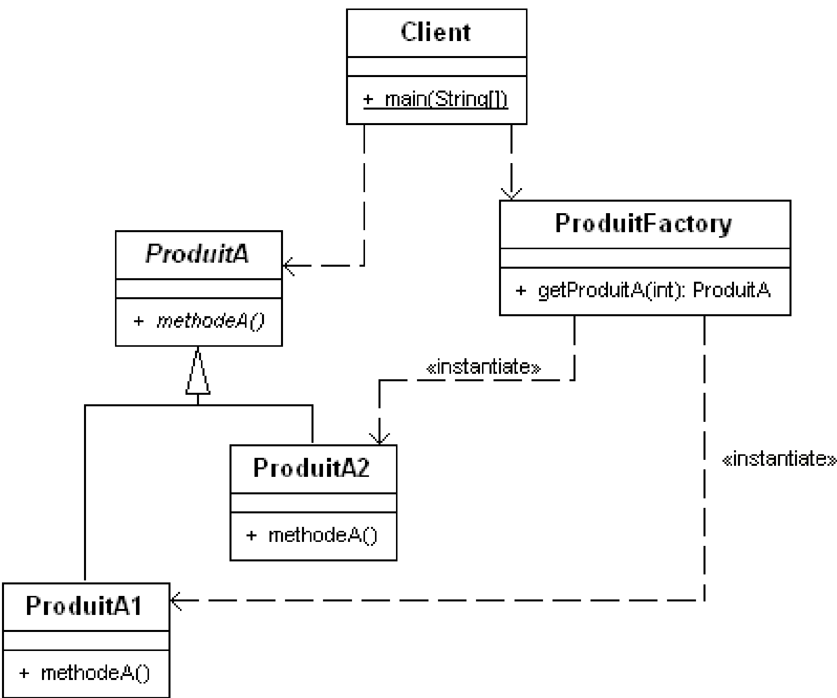
```
  public static void main(String[] args) {
    ProduitFactory produitFactory1 = new ProduitFactory1();
    ProduitFactory produitFactory2 = new ProduitFactory2();

    ProduitA produitA = null;

    System.out.println("Utilisation de la premiere fabrique");
    produitA = produitFactory1.getProduitA();
    produitA.methodeA();

    System.out.println("Utilisation de la seconde fabrique");
    produitA = produitFactory2.getProduitA();
    produitA.methodeA();

  }

}

package com.jmd.test.dej.factory1;

public abstract class ProduitFactory {

  public ProduitA getProduitA() {
    return createProduitA();
  }

  protected abstract ProduitA createProduitA();
}

package com.jmd.test.dej.factory1;

public class ProduitFactory1 extends ProduitFactory {

  protected ProduitA createProduitA() {
    return new ProduitA1();
  }
}

package com.jmd.test.dej.factory1;

public class ProduitFactory2 extends ProduitFactory {

  protected ProduitA createProduitA() {
    return new ProduitA2();
  }
}

package com.jmd.test.dej.factory1;

public abstract class ProduitA {

  public abstract void methodeA();
}

package com.jmd.test.dej.factory1;

public class ProduitA1 extends ProduitA {

  public void methodeA() {
    System.out.println("ProduitA1.methodeA()");
  }
}

package com.jmd.test.dej.factory1;

public class ProduitA2 extends ProduitA {

  public void methodeA() {
    System.out.println("ProduitA2.methodeA()");
  }
}
```

Résultat :

```
Utilisation de la premiere fabrique
ProduitA1.methodeA()
Utilisation de la seconde fabrique
ProduitA2.methodeA()
```

Il est possible d'implémenter la fabrique sous la forme d'une classe qui possède une méthode chargée de renvoyer l'instance voulue. La création de cette instance est alors réalisée en fonction de données du contexte (valeurs fournies en paramètres de la méthode, fichier de configuration, paramètres de l'application, ...).

Dans l'exemple ci-dessous, la méthode getProduitA() attend en paramètre une constante qui précise le type d'instance à créer.



Exemple : le code sources des différentes classes

```
package com.jmd.test.dej.factory2;

public class Client {

  public static void main(String[] args) {
    ProduitFactory produitFactory = new ProduitFactory();

    ProduitA produitA = null;

    produitA = produitFactory.getProduitA(ProduitFactory.TYPE_PRODUITA1);
    produitA.methodeA();

    produitA = produitFactory.getProduitA(ProduitFactory.TYPE_PRODUITA2);
    produitA.methodeA();

    produitA = produitFactory.getProduitA(3);
    produitA.methodeA();

  }

}

package com.jmd.test.dej.factory2;

public class ProduitFactory {
```

```
  public static final int TYPE_PRODUITA1 = 1;
  public static final int TYPE_PRODUITA2 = 2;

  public ProduitA getProduitA(int typeProduit) {
    ProduitA produitA = null;

    switch (typeProduit) {
      case TYPE_PRODUITA1:
        produitA = new ProduitA1();
        break;
      case TYPE_PRODUITA2:
        produitA = new ProduitA2();
        break;
      default:
        throw new IllegalArgumentException("Type de produit inconnu");
    }

    return produitA;
  }

}

package com.jmd.test.dej.factory2;

public abstract class ProduitA {

  public abstract void methodeA();
}

package com.jmd.test.dej.factory2;

public class ProduitA1 extends ProduitA {

  public void methodeA() {
    System.out.println("ProduitA1.methodeA()");
  }
}

package com.jmd.test.dej.factory2;

public class ProduitA2 extends ProduitA {

  public void methodeA() {
    System.out.println("ProduitA2.methodeA()");
  }
}
```

Résultat :

```
ProduitA1.methodeA()
ProduitA2.methodeA()
java.lang.IllegalArgumentException: Type de produit inconnu
        at com.jmd.test.dej.factory2.ProduitFactory.getProduitA(ProduitFactory.java:19)
        at com.jmd.test.dej.factory2.Client.main(Client.java:16)
Exception in thread "main"
```

Cette implémentation est plus légère à mettre en oeuvre.

Remarque : c'est une bonne pratique de toujours respecter la même convention de nommage dans le nom des fabriques et dans le nom de la méthode qui renvoie l'instance.

74.1.2. Fabrique abstraite (abstract Factory)

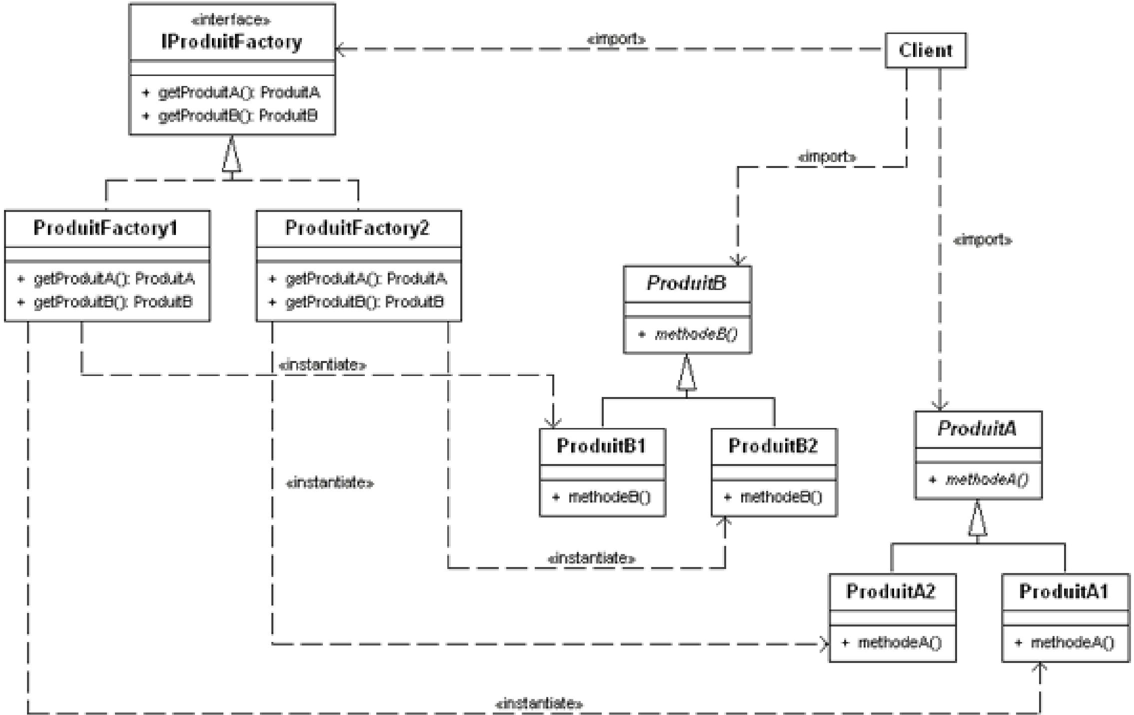
Le motif de conception Abstract Factory (fabrique abstraite) permet de fournir une interface unique pour instancier des objets d'une même famille sans avoir à connaître les classes à instancier.

L'utilisation de ce motif est pertinent lorsque :

* Le système doit être indépendant de la création des objets qu'il utilise
* Le système doit être capable de créer des objets d'une même famille

Le principal avantage de ce motif de conception est d'isoler la création des objets retournés par la fabrique. L'utilisation d'une fabrique abstraite permet de facilement remplacer une fabrique par une autre selon les besoins.

Le motif de conception fabrique abstraite peut être interprété et mis en oeuvre de différentes façons. Le diagramme UML si dessous propose une mise en oeuvre possible avec deux familles de deux produits.



Dans cet exemple, les classes suffixées par un chiffre correspondent aux classes relatives à une famille donnée.

Les classes misent en oeuvre sont :

* IProduitFactory : interface pour les fabriques de création d'objets. Elle définit donc les méthodes nécessaires à la création des objets
* ProduitFactory1 et ProduitFactory2 : fabriques qui réalisent la création des objets
* ProduitA et ProduitB : interfaces des deux familles de produits (En Java, cela peut être une classe abstraite ou une interface)
* ProduitA1, ProduitA2, ProduitB1 et ProduitB2 : implémentations des produits des deux familles
* Client : classe qui utilise la fabrique pour obtenir des objets

C'est une des classes filles de la fabrique qui se charge de la création des objets d'une famille. Ainsi tous les objets créés doivent hériter d'une classe abstraite qui sert de modèle pour toutes les classes de la famille.

Le client utilise une implémentation concrète de la fabrique abstraite pour obtenir une instance d'un objet créé par la fabrique.

Cette instance est obligatoirement du type de la classe abstraite dont toutes les classes concrètes héritent. Ainsi des objets concrets sont retournés par la fabrique mais le client ne peut utiliser que leur interface abstraite.

Comme il n'y a pas de relation entre le client et la classe concrète retournée par la fabrique, celle-ci peut renvoyer n'importe quelle classe qui hérite de la classe abstraite.

Ceci permet facilement :

* De remplacer une classe concrète par une autre.
* D'ajouter de nouveaux types d'objets qui héritent de la classe abstraite sans modifier le code qui utilise la fabrique.

Pour prendre en compte une nouvelle famille de produit dans le code client, il suffit simplement d'utiliser la fabrique dédiée à cette famille. Le reste du code client ne change pas. Ceci est beaucoup simple que d'avoir à modifier dans le code client l'instanciation des classes concrètes concernées.

Exemple :

```
package com.jmd.test.dej.abstractfactory;

public class Client {

  public static void main(String[] args) {
    IProduitFactory produitFactory1 = new ProduitFactory1();
    IProduitFactory produitFactory2 = new ProduitFactory2();

    ProduitA produitA = null;
    ProduitB produitB = null;

    System.out.println("Utilisation de la premiere fabrique");
    produitA = produitFactory1.getProduitA();
    produitB = produitFactory1.getProduitB();
    produitA.methodeA();
    produitB.methodeB();

    System.out.println("Utilisation de la seconde fabrique");
    produitA = produitFactory2.getProduitA();
    produitB = produitFactory2.getProduitB();
    produitA.methodeA();
    produitB.methodeB();

  }

}

package com.jmd.test.dej.abstractfactory;

public interface IProduitFactory {

  public ProduitA getProduitA();
  public ProduitB getProduitB();

}


package com.jmd.test.dej.abstractfactory;

public class ProduitFactory1 implements IProduitFactory {

  public ProduitA getProduitA() {
    return new ProduitA1();
  }

  public ProduitB getProduitB() {
    return new ProduitB1();
  }
}

package com.jmd.test.dej.abstractfactory;

public class ProduitFactory2 implements IProduitFactory {

  public ProduitA getProduitA() {
    return new ProduitA2();
  }

  public ProduitB getProduitB() {
    return new ProduitB2();
  }
}

package com.jmd.test.dej.abstractfactory;

public abstract class ProduitA {
```

```
  public abstract void methodeA();
}

package com.jmd.test.dej.abstractfactory;

public class ProduitA1 extends ProduitA {

  public void methodeA() {
    System.out.println("ProduitA1.methodeA()");
  }
}

package com.jmd.test.dej.abstractfactory;

public class ProduitA2 extends ProduitA {

  public void methodeA() {
    System.out.println("ProduitA2.methodeA()");
  }
}

package com.jmd.test.dej.abstractfactory;

public abstract class ProduitB {

  public abstract void methodeB();
}

package com.jmd.test.dej.abstractfactory;

public class ProduitB1 extends ProduitB {

  public void methodeB() {
    System.out.println("ProduitB1.methodeB()");
  }
}

package com.jmd.test.dej.abstractfactory;

public class ProduitB2 extends ProduitB {

  public void methodeB() {
    System.out.println("ProduitB2.methodeB()");
  }
}
```

Résultat :

```
Utilisation de la premiere fabrique
ProduitA1.methodeA()
ProduitB1.methodeB()
Utilisation de la seconde fabrique
ProduitA2.methodeA()
ProduitB2.methodeB()
```

Une fabrique concrète est généralement un singleton.

74.1.3. Monteur (Builder)

Cette section sera développée dans une version future de ce document

74.1.4. Prototype (Prototype)

Cette section sera développée dans une version future de ce document

74.1.5. Singleton (Singleton)

Ce modèle permet de définir une classe dont il ne pourra y avoir qu'une seule instance. Le modèle assure aussi l'accès à cette unique instance.

Ce modèle est particulièrement utile pour le développement d'objets de type gestionnaire. En effet ce type d'objet doit être unique car il gère d'autres objets par exemple un gestionnaire de logs.

Pour mettre en oeuvre ce modèle, il faut :

- créer une instance de la classe et la stocker dans une variable privée
- empêcher l'utilisation du ou des constructeurs
- fournir une méthode qui renvoie l'instance stockée dans la variable privée

Exemple :

```
public class MonSingleton {

  /** Singleton. */
  private static MonSingleton monSingleton = new MonSingleton();

  /**
   * Constructeur de la classe MonSingleton.
   */
  private MonSingleton() {
    super();
  }

  /**
   * Renvoie le singleton.
   */
  public static MonSingleton get() {
    return monSingleton;
  }

  public void afficher() {
    System.out.println("Singleton");
  }
}
```

Pour vérifier que l'usage du constructeur est impossible, il suffit de compiler une classe qui tente d'en faire usage.

Exemple :

```
public class TestSingleton1 {
  public static void main() {
    MonSingleton ms = new MonSingleton();
    ms.afficher();
  }
}
```

Résultat :

```
C:\java>javac TestSingleton1.java

TestSingleton1.java:4:
No constructor matching MonSingleton() found in class Mon
Singleton.

                MonSingleton ms = new MonSingleton();
                                  ^
1 error
```

Le compilateur indique une erreur car le constructeur a été déclaré private pour empêcher son appel.

Pour pouvoir utiliser l'instance de la classe, il faut appeler la méthode qui renvoie l'instance unique.

Exemple :

```
public class TestSingleton2 {
            public static void main(String[] args) {
                        MonSingleton ms = MonSingleton.get();
                        ms.afficher();
            }
}
```

Résultat :

```
C:\java>javac TestSingleton2.java

C:\java>java TestSingleton2
Singleton

C:\java>
```

Il faut impérativement déclarer le ou les constructeurs par défaut et explicitement déclarer un constructeur par défaut pour empêcher le compilateur de l'ajouter.

L'instanciation de l'unique instance peut être réalisée de façon statique ou réalisée à la demande. Dans ce cas, la méthode qui renvoie l'instance doit vérifier qu'elle existe et dans le cas contraire, créer l'instance, la stockée dans la variable privée et la renvoyer.

Pour accentuer encore l'assurance de l'unicité de l'instance, il peut être utile de déclarer la classe finale pour éviter que celle ci soit héritée. En effet cette possibilité permettrait de créer un nouveau constructeur dans la classe fille ou de rendre celle ci clonable.

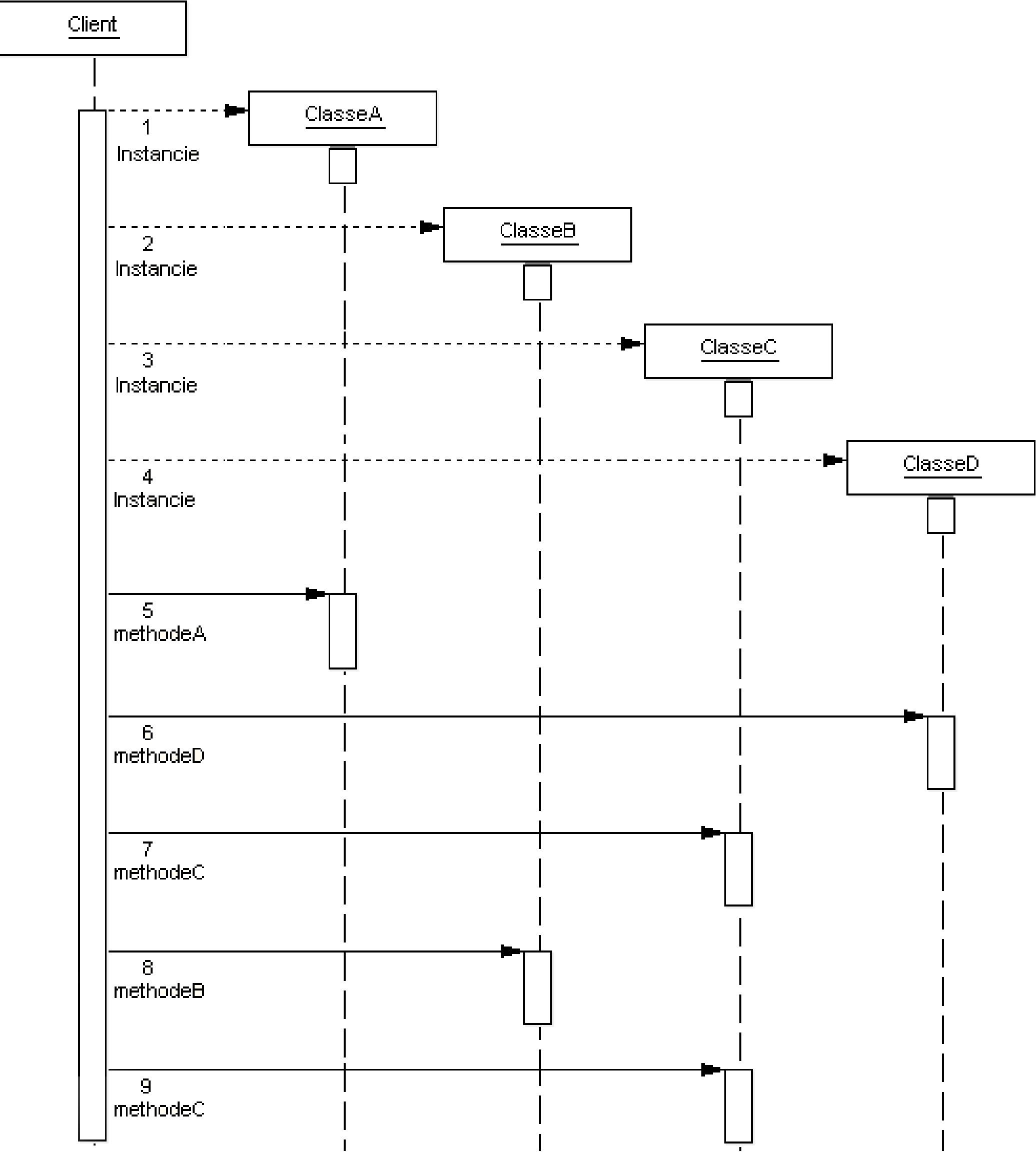
74.2. Les modèles de structuration

74.2.1. Façade (Facade)

Une bonne pratique de conception et d'essayer de limiter le couplage existant entre des fonctionnalités proposées par différentes entités. Dans la pratique, il est préférable de développer un petit nombre de classes et de proposer une classe pour les utiliser. C'est ce que propose le motif de conception façade.

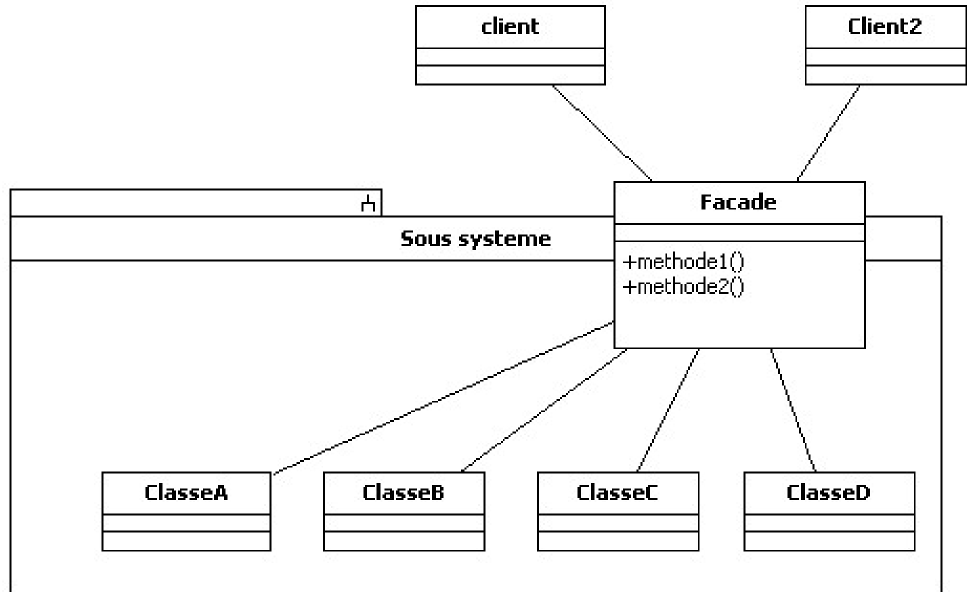
Le but est de proposer une interface facilitant la mise en oeuvre d'un ensemble de classes généralement regroupées dans un ou plusieurs sous systèmes. Il permet d'offrir un niveau d'abstraction entre l'ensemble de classes et celles qui souhaitent les utiliser en proposant une interface de plus haut niveau pour utiliser les classes du sous système.

Exemple : un client qui utilise des classes d'un sous système directement



Cet exemple volontairement simpliste va être modifié pour mettre en oeuvre le modèle de conception Façade.

Employer ce modèle aide à simplifier une grande partie de l'interface pour utiliser les classes du sous système. Il facilite la mise en oeuvre de plusieurs classes en fournissant une couche d'abstraction supplémentaire entre ce dernier et les classes qui les utilisent. Le modèle Façade permet donc de faciliter la compréhension et l'utilisation d'un sous système complexe que ce soit pour faciliter l'utilisation de tout ou partie du système ou pour forcer une utilisation particulière du système.



Les classes du sous système encapsulent les traitements qui seront exécutés par des appels de méthodes de l'objet Façade. Ces classes ne doivent pas connaître ni de surcroît avoir de référence sur l'objet Façade.

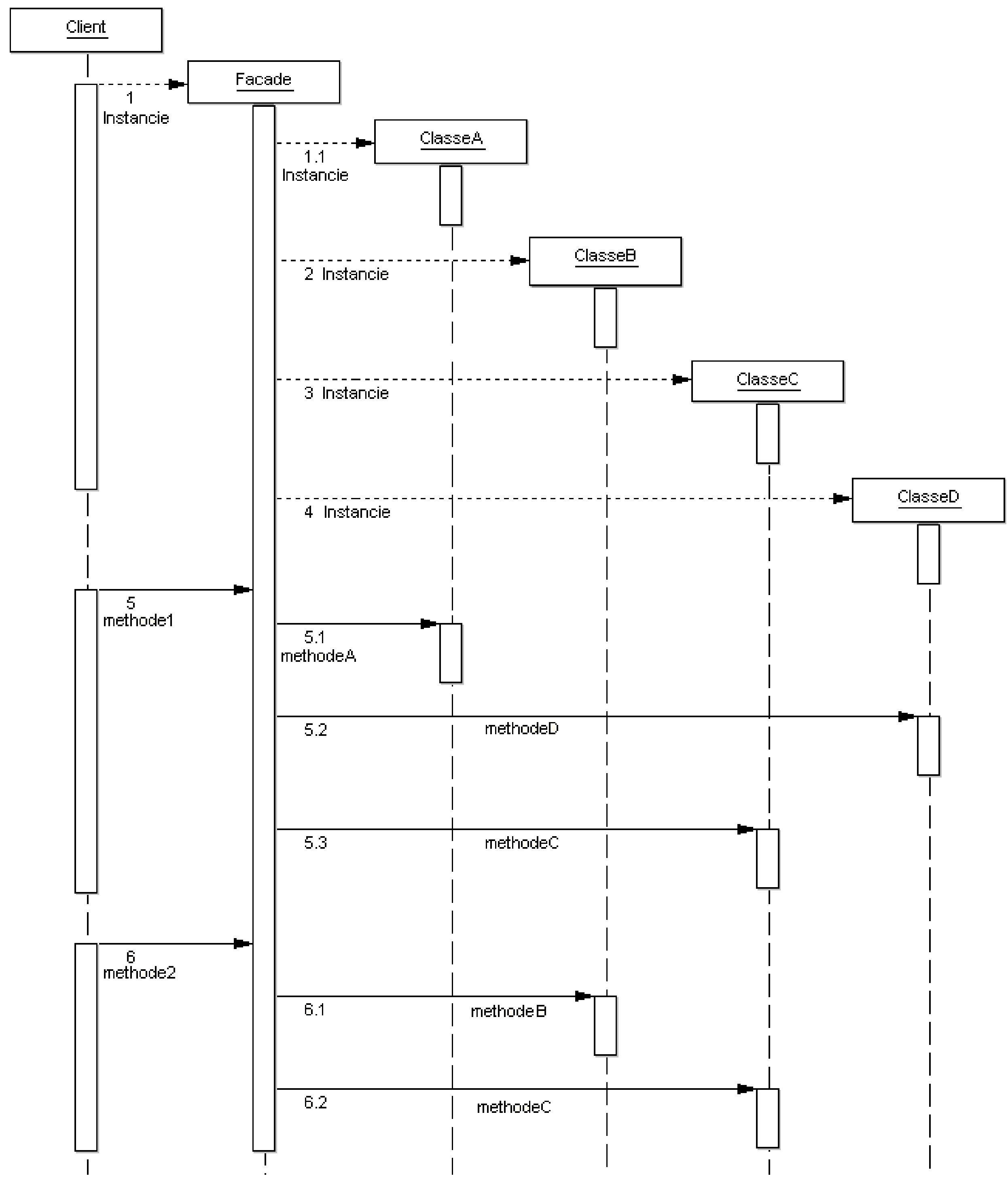
La façade propose un ensemble de méthodes qui vont réaliser les appels nécessaires aux classes du sous système pour offrir des fonctionnalités cohérentes. Elle propose une interface pour faciliter l'utilisation du sous système en implémentant les traitements requis pour utiliser les classes du sous système.

La classe qui implémente le modèle Façade encapsule les appels aux différentes classes nécessaires à l'exécution d'un traitement cohérent impliquant plusieurs de ces classes. Il fait donc office de point d'entrée pour utiliser le sous système.

Ce modèle implique donc plusieurs classes :

- Le client qui va utiliser la façade
- La façade
- Les classes du sous système utilisées par la façade

Exemple :



Le code à utiliser dans la classe client est réduit ce qui va en faciliter la maintenance. La façade masque donc les complexités du sous système utilisé et fournit une interface simple d'accès pour les clients qui l'utilise.

Exemple :

```
public class ClientTestFacade {
  public static void main(String[] argv) {
    TestFacade facade = new TestFacade();

    facade.methode1();
    facade.methode2();
  }
}

public class TestFacade {

  ClasseA classeA;
  ClasseB classeB;
```

```
  ClasseC classeC;
  ClasseD classeD;

  public TestFacade() {
    classeA = new ClasseA();
    classeB = new ClasseB();
    classeC = new ClasseC();
    classeD = new ClasseD();
  }

  public void methode1() {
    System.out.println("Methode2 : ");
    classeA.methodeA();
    classeD.methodeD();
    classeC.methodeC();
  }

  public void methode2() {
    System.out.println("Methode1 : ");
    classeB.methodeB();
    classeC.methodeC();
  }

}

public class ClasseA {
  public void methodeA() {
    System.out.println(" - MethodeA ClasseA");
  }
}

public class ClasseB {
  public void methodeB() {
    System.out.println(" - MethodeB Classe B");
  }
}

public class ClasseC {
  public void methodeC() {
    System.out.println(" - MethodeC ClasseC");
  }
}

public class ClasseD {
  public void methodeD() {
    System.out.println(" - MethodeD ClasseD");
  }
}
```

Résultat :

```
Methode2 :
 - MethodeA ClasseA
 - MethodeD ClasseD
 - MethodeC ClasseC
Methode1 :
 - MethodeB Classe B
 - MethodeC ClasseC
```

Le modèle Façade peut être utilisé pour :

- Faciliter l'utilisation partielle d'un sous système complexe ou de plusieurs classes
- Masquer l'existence d'un sous système
- Ajouter des fonctionnalités sans modifier le sous système
- Assurer un découplage entre le client et le sous système (par exemple pour chaque couche d'une architecture logiciel N tiers)

L'utilisation d'une façade permet au client de limiter le nombre d'objets à utiliser puisqu'il se contente simplement d'appeler une ou plusieurs méthodes de la façade. C'est ces méthodes qui vont utiliser les classes du sous système,

masquant ainsi au client toute la complexité de leur mise en oeuvre.

Il peut être pratique de définir une façade sans état (aucune des méthodes de la façade n'utilisent des membres non statiques de la classe) car dans ce cas, une seule et unique instance de la façade peut être définie côté client en mettant en oeuvre le modèle de conception singleton prévu à cet effet.

Il est possible de proposer des fonctionnalités supplémentaires dans la façade qui enrichisse la mise en oeuvre du sous système.

La façade peut aussi être utilisée pour masquer le sous système. La façade peut encapsuler les classes du sous système et ainsi cacher au client l'existence du sous système. Cette mise en oeuvre facilite le remplacement du sous système par un autre : il suffit simplement de modifier la façade pour que le client continue à fonctionner.

Il est possible que toutes les fonctionnalités proposées par les classes du sous système ne soient pas accessible via la façade : son but est de simplifier leurs utilisations mais pas de proposer toutes les fonctionnalités.

Ce motif de conception est largement utilisé.

74.2.2. Décorateur (Decorator)

Le motif de conception décorateur (decorator en anglais) permet d'ajouter des fonctionnalités à un objet en mettant en oeuvre une solution plus souple que l'héritage : il permet d'ajouter des fonctionnalités à une ou plusieurs méthodes existantes d'une classe dynamiquement.

La programmation orienté objet propose l'héritage pour ajouter des fonctionnalités à une classe, cependant l'héritage possède quelques contraintes et il n'est toujours possible de le mettre en oeuvre (par exemple si la classe est finale).

Avec l'héritage, il serait nécessaire de définir autant de classe fille que de cas ce qui peut vite devenir ingérable. Avec l'utilisation d'un décorateur, il suffit de définir un décorateur pour chaque fonctionnalité et de les utiliser par combinaison en fonction des besoins. L'héritage ajoute des fonctionnalités de façon statique (à la compilation) alors que le décorateur ajoute des fonctionnalités de façon dynamique (à l'exécution).

L'héritage crée une nouvelle classe qui reprend les fonctionnalités de la classe mère et les modifie ou les enrichie. Mais il possède quelques inconvénients :

- Il n'est pas toujours possible (par exemple pour une classe déclarée finale)
- Cela peut faire augmenter le nombre de classes pour définir tous les cas de figure requis
- L'ajout des fonctionnalités est statique

Le modèle de conception décorateur apporte une solution à ces trois inconvénients et propose donc une alternative à l'héritage.

Le motif de conception décorateur permet de définir un ensemble de classes possédant une base commune mais proposant chacune des variantes sans utiliser l'héritage qui est le mécanisme proposé de base par la programmation orienté objet. Ceci permet d'enrichir une classe avec des fonctionnalités supplémentaires.

Ce motif est dédié à la création de variantes d'une classe et facilite ainsi le développement de plusieurs classes plutôt qu'une seule classe qui prennent en compte les variantes. Ce motif permet aussi de réaliser des combinaisons de plusieurs variantes.

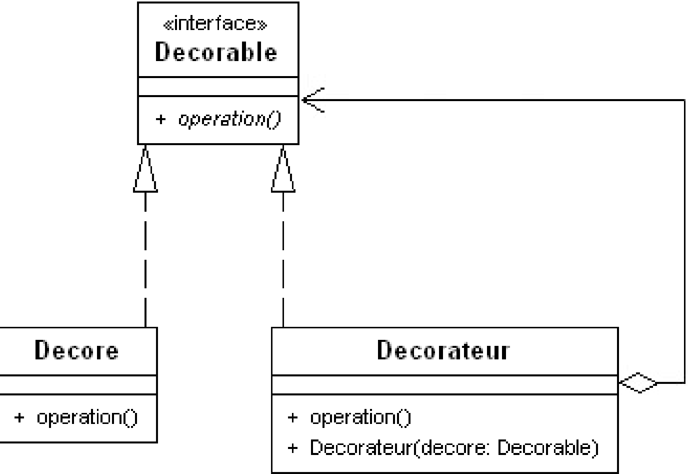
Ce motif de conception est donc généralement utilisé lorsqu'il n'est pas possible de prédéfinir le nombre de combinaison induite par l'ajout de nombreuses fonctionnalités ou si ce nombre est trop important. Le principe du motif de conception décorateur est d'utiliser la composition : le décorateur contient un objet décoré. L'appel d'une méthode du décorateur provoque l'exécution de la méthode correspondante du décoré et des fonctionnalités ajoutées par le décorateur.

Le motif décorateur repose sur deux entités :

- Le décoré : interface ou classe qui définit les fonctionnalités de base
- Le décorateur : classe enrichie qui contient les fonctionnalités de base plus celles ajoutées

Le décorateur encapsule le décoré dont l'instance est généralement fournie dans les paramètres d'un constructeur. Il est important que l'interface du décorateur reprenne celle de l'objet décoré.

Pour permettre de combiner les décorations, le décoré et le décorateur doivent implémenter une interface commune.

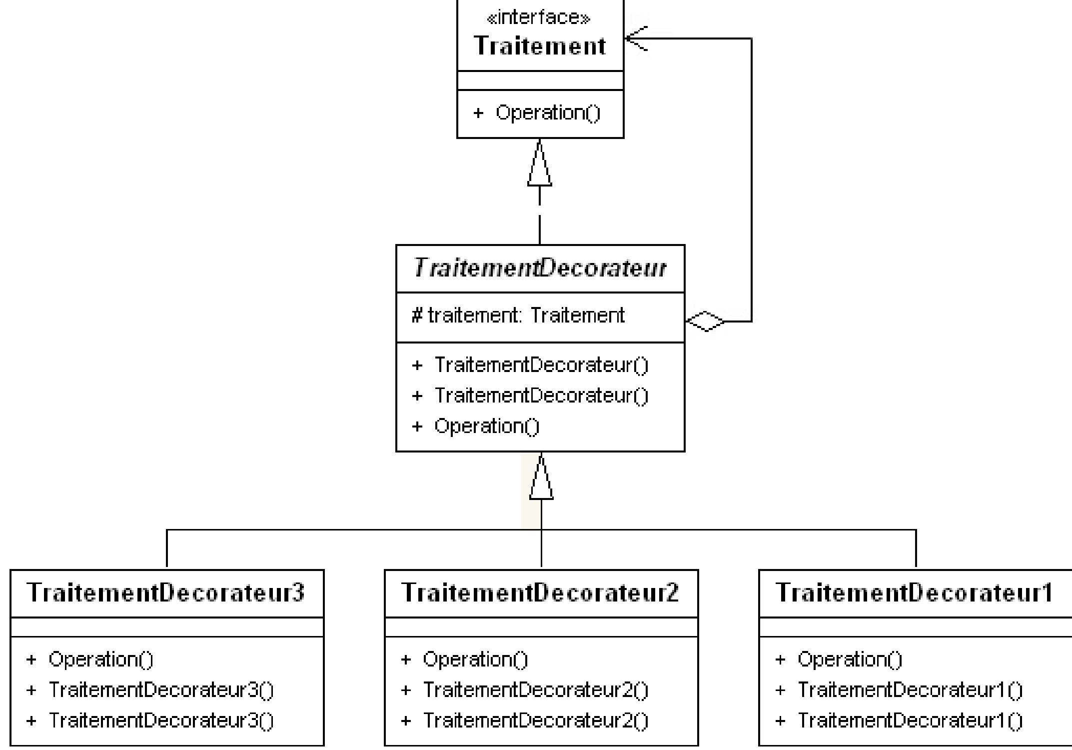


La combinaison peut alors être répétée pour construire un objet qui va contenir les différentes fonctionnalités proposées par les décorateurs utilisés

Le motif de conception décorateur est particulièrement utile dans plusieurs cas :

- Définition de fonctionnalités génériques qui peuvent prendre plusieurs formes
- Définition de plusieurs fonctionnalités optionnelles

Il permet de créer un objet qui va être composé des fonctionnalités requises par ajout successif des différents décorateurs qui proposent les fonctionnalités requises.



Un des avantages de ce motif de conception est de n'avoir à créer qu'une seule classe pour proposer des fonctionnalités supplémentaires aux classes qui mettent en oeuvre ce motif. Avec l'héritage, il serait nécessaire de créer autant de classes filles que de classes concernées ou de gérer la fonctionnalité dans une classe mère en modifiant cette dernière pour prendre en compte cet ajout avec tous les risques que cela peut engendrer.

Il faut définir une interface qui va déclarer toutes les fonctionnalités des décorés.

Exemple : interface Traitement

```
package com.jmdoudoux.test.dp.decorateur;

public interface Traitement {
  public void Operation();
}
```

Il faut définir un décorateur de base qui implémente l'interface et possède une référence sur une instance de l'interface. Cette référence est le décoré qui va être enrichi des fonctionnalités du décorateur.

Exemple : classe abstraite TraitementDecorateur

```
package com.jmdoudoux.test.dp.decorateur;

public abstract class TraitementDecorateur implements Traitement {

  protected Traitement traitement;

  public TraitementDecorateur()
  {
  }

  public TraitementDecorateur(Traitement traitement)
  {
    this.traitement = traitement;
  }

  public void Operation() {
    if (traitement != null)
    {
      traitement.Operation();
    }
  }
}
```

Il faut définir les décorateurs qui héritent du décorateur de base et implémentent les fonctionnalités supplémentaires qu'ils sont chargés de proposer.

Il faut définit chacun des décorateurs.

Exemple : TraitementDecorateur1

```
package com.jmdoudoux.test.dp.decorateur;

public class TraitementDecorateur1 extends TraitementDecorateur {

  public TraitementDecorateur1() {
    super();
  }

  public TraitementDecorateur1(Traitement traitement) {
    super(traitement);
  }

  @Override
  public void Operation() {
    if (traitement != null)
    {
      traitement.Operation();
```

```
    }
    System.out.println("TraitementDecorateur1.Operation()");
  }

}
```

Exemple : TraitementDecorateur2

```
package com.jmdoudoux.test.dp.decorateur;

public class TraitementDecorateur2 extends TraitementDecorateur {

  public TraitementDecorateur2() {
    super();
  }

  public TraitementDecorateur2(Traitement traitement) {
    super(traitement);
  }

  @Override
  public void Operation() {
    if (traitement != null)
    {
      traitement.Operation();
    }

    System.out.println("TraitementDecorateur2.Operation()");
  }

}
```

Exemple : TraitementDecorateur3

```
package com.jmdoudoux.test.dp.decorateur;

public class TraitementDecorateur3 extends TraitementDecorateur {

  public TraitementDecorateur3() {
    super();
  }

  public TraitementDecorateur3(Traitement traitement) {
    super(traitement);
  }

  @Override
  public void Operation() {
    if (traitement != null)
    {
      traitement.Operation();
    }
    System.out.println("TraitementDecorateur3.Operation()");
  }

}
```

Il est possible de fournir une classe d'implémentation par défaut.

Il est pratique d'utiliser le motif de conception fabrique pour construire l'objet décoré finale. Dans ce cas, une implémentation par défaut de l'interface peut être utile.

Exemple : TraitementTest.java

```
package com.jmdoudoux.test.dp.decorateur;

public class TraitementTest {

  public static void main(String[] args) {
```

```
    System.out.println("traitement 1 2 3");
    Traitement traitement123 = new TraitementDecorateur3(
      new TraitementDecorateur2(new TraitementDecorateur1()));
    traitement123.Operation();

    System.out.println("traitement 1 3");
    Traitement traitement13 = new TraitementDecorateur3(new TraitementDecorateur1());
    traitement13.Operation();
  }

}
```

Résultat d'exécution :

```
traitement 1 2 3
TraitementDecorateur1.Operation()
TraitementDecorateur2.Operation()
TraitementDecorateur3.Operation()
traitement 1 3
TraitementDecorateur1.Operation()
TraitementDecorateur3.Operation()
```

L'API de base de Java utilise le motif de conception décorateur notamment dans l'API IO

74.3. Les modèles de comportement

La suite de ce chapitre sera développée dans une version future de ce document

75. Des normes de développement

Le but de ce chapitre est de proposer un ensemble de conventions et de règles pour faciliter la compréhension et donc la maintenance du code.

Ces règles ne sont pas à suivre explicitement à la lettre : elles sont uniquement présentées pour inciter le ou les développeurs à définir et à utiliser des règles dans la réalisation de son code surtout dans le cadre d'un travail en équipe. Les règles proposées sont celles couramment utilisées. Il n'existe cependant pas de règles absolues et chacun pourra utiliser tout ou partie des règles proposées.

La définition de conventions et de règles est importante pour plusieurs raisons :

- La majorité du temps passé à coder est consacré à la maintenance évolutive et corrective d'une application
- Ce n'est pas toujours voir rarement l'auteur du code qui effectue ces maintenances
- ces règles facilitent la lisibilité et donc la compréhension du code

Le contenu de ce document est largement inspiré par les conventions de codage proposées par Sun à l'URL suivante : http://java.sun.com/docs/codeconv/index.html

Ce chapitre contient plusieurs sections :

- Les fichiers
- La documentation du code
- Les déclarations
- Les séparateurs
- Les traitements
- Les règles de programmation

75.1. Les fichiers

Java utilise des fichiers pour stocker les sources et le byte code des classes.

75.1.1. Les packages

Les packages permettent de grouper les classes sous une forme hiérarchisée. Le choix des critères de regroupement est laissé aux développeurs.

Il est préférable de regrouper les classes par packages selon des critères fonctionnels.

Les fichiers inclus dans un package doivent être insérer dans une arborescence de répertoires équivalentes.

75.1.2. Le nom de fichiers

Chaque fichier source ne doit contenir qu'une seule classe ou interface publique. Le nom du fichier doit être identique au nom de cette classe ou interface publique en respectant la casse.

Il faut éviter dans ce nom d'utiliser des caractères accentués qui ne sont pas toujours utilisables par tous les systèmes d'exploitation.

Les fichiers sources ont pour extension .java car le compilateur javac fourni avec le J.D.K. utilise cette extension

Exemple :

```
javac MaClasse.java
```

Les fichiers binaires contenant le byte-code ont pour extension .class car le compilateur génère un fichier avec cette extension à partir du fichier source .java correspondant. De plus, elle est obligatoire pour l'interpréteur Java qui l'ajoute automatiquement au nom du fichier fourni en paramètre.

Exemple :

```
java MaClasse
```

75.1.3. Le contenu des fichiers sources

Un fichier ne devrait pas contenir plus de 2 000 lignes de code.

Des interfaces ou classes privées ayant une relation avec la classe publique peuvent être rassemblées dans un même fichier. Dans ce cas, la classe publique doit être la première dans le fichier.

Chaque fichier source devrait contenir dans l'ordre :

1. un commentaire concernant le fichier
2. les clauses concernant la gestion des packages (la déclaration et les importations)
3. les déclarations de classes ou de l'interface

75.1.4. Les commentaires de début de fichier

Chaque fichier source devrait commencer par un commentaire multi-lignes contenant au minimum des informations sur le nom de la classe, la version, la date, éventuellement le copyright et tous les autres commentaires utiles :

Exemple :

```
/*
 * Nom de classe : MaClasse
 *
 * Description   : description de la classe et de son rôle
 *
 * Version       : 1.0
 *
 * Date          : 23/02/2001
 *
 * Copyright     : moi
 */
```

75.1.5. Les clauses concernant les packages.

La première ligne de code du fichier devrait être une clause package indiquant à quel paquetage appartient la classe. Le fichier source doit obligatoirement être inclus dans une arborescence correspondante au nom du package.

Il faut indiquer ensuite l'ensemble des paquetages à importer : ceux dont les classes vont être utilisées dans le code.

Exemple :

```
package monpackage;

import java.util.*;
import java.text.*;
```

75.1.6. La déclaration des classes et des interfaces

Les différents éléments qui composent la définition de la classe ou de l'interface devraient être indiqués dans l'ordre suivant :

1. les commentaires au format javadoc de la classe ou de l'interface
2. la déclaration de la classe ou de l'interface
3. les variables de classes (déclarées avec le mot clé static) triées par ordre d'accessibilité : d'abord les variables déclarées public, protected, package friendly (sans modificateur d'accès) et enfin private
4. les variables d'instances triées par ordre d'accessibilité : d'abord les variables déclarées public, protected, package friendly (sans modificateur d'accès) et enfin private
5. le ou les constructeurs
6. les méthodes : elles seront regroupées par fonctionnalités plutôt que selon leur accessibilité

75.2. La documentation du code

Il existe deux types de commentaires en java :

- les commentaires de documentation : ils permettent en respectant quelques règles d'utiliser l'outil javadoc fourni avec le J.D.K. qui formate une documentation des classes, indépendante de l'implémentation du code,
- les commentaires de traitements : ils fournissent un complément d'information dans le code lui-même.

Les commentaires ne doivent pas être entourés par de grands cadres dessinés avec des étoiles ou d'autres caractères.

Les commentaires ne devraient pas contenir de caractères spéciaux tels que le saut de page.

75.2.1. Les commentaires de documentation

Les commentaires de documentation utilisent une syntaxe particulière utilisée par l'outil javadoc de Sun pour produire une documentation standardisée des classes et interfaces au format HTML. La documentation de l'API du J.D.K. est le résultat de l'utilisation de cet outil de documentation

75.2.1.1. L'utilisation des commentaires de documentation

Cette documentation concerne les classes, les interfaces, les constructeurs, les méthodes et les champs.

La documentation est définie entre les caractères /** et */ selon le format suivant :

Exemple :

```
/**
 *  Description de la methode
 */
public void maMethode() {
```

La première ligne de commentaires ne doit contenir que /**

Les lignes de commentaires suivantes doivent obligatoirement commencer par un espace et une étoile. Toutes les premières étoiles doivent être alignées.

La dernière ligne de commentaires ne doit contenir que */ précédé d'un espace.

Un tel commentaire doit être défini pour chaque entité : une classe, une interface et chaque membre (variables et méthodes).

Javadoc définit un certain nombre de tags qu'il est possible d'utiliser pour apporter des précisions sur plusieurs informations.

Ces tags permettent de définir des caractéristiques normalisées. Il est possible d'inclure dans les commentaires des tags HTML de mise en forme (PRE, TT, EM ...) mais il n'est pas recommandé d'utiliser des tags HTML de structure tel que Hn, HR, TABLE ... qui sont utilisés par javadoc pour formater la documentation

Il faut obligatoirement faire précéder l'entité documentée par son commentaire car l'outil associe la documentation à la déclaration de l'entité qui la suit.

75.2.1.2. Les commentaires pour une classe ou une interface

Pour les classes ou interfaces, javadoc définit les tags suivants : @see, @version, @author, @copyright, @security, @date, @revision, @note

Les tags @copyright, @security, @date, @revision et @note ne sont pas traités par javadoc.

Exemple :

```
/**
 * NomClasse - description de la classe
 * explication supplémentaire si nécessaire
 *
 * @version1.0
 *
 * @see UneAutreClasse
 * @author Jean Michel D.
 * @copyright (C) moi 2001
 * @date 01/09/2000
 * @notes notes particulières sur la classe
 *
 * @revision référence
 *         date 15/11/2000
 *         author Michel M.
 *         raison  description
 *         description supplémentaire
 */
```

75.2.1.3. Les commentaires pour une variable de classe ou d'instance

75.2.1.4. Les commentaires pour une méthode

Pour les méthodes, javadoc définit les tags suivants : @see, @param, @return, @exception, @author, @note

Le tag @note n'est pas traité par javadoc.

Exemple :

```
/**
 * nomMethode - description de la méthode
 *              explication supplémentaire si nécessaire
 *
 *     exemple d'appel de la methode
 * @return      description de la valeur de retour
 * @param       arg1 description du 1er argument
 *    :            :          :
 * @param       argN description du Neme argument
 * @exception   Exception1  description de la première exception
 *    :            :          :
 * @exception ExceptionN  description de la Neme exception
 *
 * @see UneAutreClasse#UneAutreMethode
 * @author   Jean Dupond
 * @date     12/02/2001
 * @note     notes particulières.
 */
```

Remarques :

- @return ne doit pas être utilisé avec les constructeurs et les méthodes sans valeur de retour (void)
- @param ne doit pas être utilisé s'il n'y a pas de paramètres
- @exception ne doit pas être utilisé si il n'y pas d'exception propagée par la méthode
- @author doit être omis si il est identique à celui du tag @author de la classe
- @note ne doit pas être utilisé s'il n'y a pas de note

75.2.2. Les commentaires de traitements

Ces commentaires doivent ajouter du sens et des précisions au code : ils ne doivent pas reprendre ce que le code exprime mais expliquer clairement son rôle.

Tous les commentaires utiles à une meilleure compréhension du code et non inclus dans les commentaires de documentation seront insérés avec des commentaires de traitements. Il existe plusieurs styles de commentaires :

- les commentaires sur une ligne
- les commentaires sur une portion de ligne
- les commentaires multi-lignes

Il est conseillé de mettre un espace après le délimiteur de début de commentaires et avant le délimiteur de fin de commentaires lorsqu'il y en a un, afin d'améliorer sa lisibilité.

75.2.2.1. Les commentaires sur une ligne

Ces commentaires sont définis entre les caractères /* et */ sur une même ligne

Exemple :

```
if (i < 10) {
   /* commentaires utiles au code */
   ...
}
```

Ce type de commentaires doit être précédé d'une ligne blanche et doit suivre le niveau d'indentation courant.

75.2.2.2. Les commentaires sur une portion de ligne

Ce type de commentaires peut apparaître sur la ligne de code qu'elle commente mais il faut inclure un espace conséquent qui permette de séparer le code et le commentaire.

Exemple :

```
i++;                        /* commentaires utiles au code */
```

Si plusieurs lignes qui se suivent contiennent chacune un tel commentaire, il faut les aligner :

Exemple :

```
i++;                        /* commentaires utiles au code */
j++;                        /* second commentaires utiles au code */
```

75.2.2.3. Les commentaires multi-lignes

Exemple :

```
/*
 * Commentaires utiles au code
 */
```

Ce type de commentaires doit être précédé d'une ligne blanche et doit suivre le niveau d'indentation courant.

75.2.2.4. Les commentaires de fin de ligne

Ce type de commentaire peut délimiter un commentaire sur une ligne complète ou une fin de ligne.

Exemple :

```
i++;          // commentaires utiles au code
```

Ce type de commentaires peut apparaître sur la ligne de code qu'elle commente mais il faut inclure un espace conséquent qui permette de séparer le code et le commentaire.

Si plusieurs lignes qui se suivent contiennent chacune un tel commentaire, il faut les aligner :

Exemple :

```
i++;                        // commentaires utiles au code
j++;                        // second commentaires utiles au code
```

L'usage de cette forme de commentaires est fortement recommandé car il est possible d'inclure celui ci dans un autre de la forme /* */ et ainsi mettre en commentaire un morceau de code incluant déjà des commentaires.

75.3. Les déclarations

75.3.1. La déclaration des variables

Il n'est pas recommandé d'utiliser des caractères accentués dans les identifiants de variables, cela peut éventuellement poser des problèmes dans le cas ou le code est édité sur des systèmes d'exploitation qui ne les gèrent pas correctement.

Il ne doit y avoir qu'une seule déclaration d'entité par ligne.

Exemple :

```
String nom;
String prenom;
```

Cet exemple est préférable à

Exemple :

```
String nom, prenom;          //ce type de déclaration n'est pas recommandée
```

Il faut éviter de déclarer des variables de types différents sur la même ligne même si cela est accepté par le compilateur.

Exemple :

```
int age, notes[];            // ce type de déclaration est à éviter
```

Il est préférable d'aligner le type, l'identifiant de l'objet et les commentaires si plusieurs déclarations se suivent pour retrouver plus facilement les divers éléments.

Exemple :

```
String          nom          //nom de l'eleve
String          prenom       //prenom de l'eleve
int             notes[]      //notes de l'eleve
```

Il est fortement recommandé d'initialiser les variables au moment de leur déclaration.

Il est préférable de rassembler toutes les déclarations d'un bloc au début de ce bloc. (un bloc est un morceau de code entouré par des accolades).

La seule exception concerne la déclaration de la variable utilisée comme index dans une boucle.

Exemple :

```
for (int i = 0 ; i < 9 ; i++) { ... }
```

Il faut proscrire la déclaration d'une variable qui masque une variable définie dans un bloc parent afin de ne pas complexifier inutilement le code.

Exemple :

```
int taille;
...
```

```
void maMethode() {
    int taille;
}
```

75.3.2. La déclaration des classes et des méthodes

Il ne doit pas y avoir d'espaces entre le nom d'une méthode et sa parenthèse ouvrante.

L'accolade ouvrante qui définit le début du bloc de code doit être à la fin de la ligne de déclaration.

L'accolade fermante doit être sur une ligne séparée dont le niveau d'indentation correspond à celui de la déclaration.

Une exception tolérée concerne un bloc de code vide : dans ce cas les deux accolades peuvent être sur la même ligne.

La déclaration d'une méthode est précédée d'une ligne blanche.

Exemple :

```
class MaClasse extends MaClasseMere {
   String nom;
   String prenom;
   MaClasse(String nom, String prenom) {
      this.nom = nom;
      this.prenom = prenom;
   }
   void neRienFaire() {}
}
```

Il faut éviter d'écrire des méthodes longues et compliquées : le traitement réalisé par une méthode doit être simple et fonctionnel. Cela permet d'écrire des méthodes réutilisables dans la classe et facilite la maintenance. Cela permet aussi d'éviter la redondance de code.

Java propose deux syntaxes pour déclarer une méthode qui retourne un tableau : la première syntaxe est préférable.

Exemple :

```
public int[] notes() {      // utiliser cette forme
public int notes()[] {
```

Il est fortement recommandé de toujours initialiser les variables locales d'une méthode lors de leur déclaration car contrairement aux variables d'instances, elles ne sont pas implicitement initialisées avec une valeur par défaut selon leur type.

75.3.3. La déclaration des constructeurs

Elle suit les mêmes règles que celles utilisées pour les méthodes.

Il est préférable de définir explicitement le constructeur par défaut (le constructeur sans paramètre). Soit le constructeur par défaut est fourni par le compilateur et dans ce cas il est préférable de le définir soit il existe d'autres constructeurs et dans ce cas le compilateur ne définit pas de constructeur par défaut.

Il est préférable de toujours initialiser les variables d'instance dans un constructeur soit avec les valeurs fournies en paramètres du constructeur soit avec des valeurs par défaut.

Exemple :

```
class Personne {
   String nom;
   String prenom;
   int    age;

   Personne() {
      this( "Inconnu", "inconnu", -1 );
   }

   Personne( String nom, String prenom, int age ) {
      this.name    = nom;
      this.address = prenom;
      this.age     = age;
   }
}
```

Il est possible d'appeler un constructeur dans un autre constructeur pour faciliter l'écriture.

Il est recommandé de toujours appeler explicitement le constructeur hérité lors de la redéfinition d'un constructeur dans une classe fille grâce à l'utilisation du mot clé super.

Exemple :

```
class Employe extends Personne {
   int matricule;
   Employee() {
      super();
      matricule = -1;
   }

   Employee(String nom, String prenom, int age, int matricule) {
      super(nom, prenom, age);
      this.matricule = matricule;
   }
}
```

Il est conseillé de ne mettre que du code d'initialisation des variables d'instances dans un constructeur et de mettre les traitements dans des méthodes qui seront appelées après la création de l'objet.

75.3.4. Les conventions de nommage des entités

Les conventions de nommage des entités permettent de rendre les programmes plus lisibles et plus faciles à comprendre. Ces conventions permettent notamment de déterminer rapidement quelle entité désigne un identifiant, une classe ou une méthode.

Entités	Règles	Exemple
Les packages	Toujours écrits tout en minuscules (norme java 1.2)	com.entreprise.projet
Les classes, les interfaces et les constructeurs	La première lettre est en majuscule. Si le nom est composé de plusieurs mots, la première lettre de chaque mot doit être en majuscule, ne pas mettre de caractère underscore '_' Le nom d'une classe peut finir par impl pour la distinguer d'une interface qu'elle implémente. Les classes qui définissent des exceptions doivent finir par Exception.	MaClasse MonInterface MaClasse()
Les méthodes	Leur nom devrait contenir un verbe. La première lettre est obligatoirement une	public float calculerMontant() {

	minuscule. Si le nom est composé de plusieurs mots, la première lettre de chaque mot doit être en majuscule sans mettre de caractère underscore '_' Les méthodes pour obtenir la valeur d'un champ doivent commencer par get suivi du nom du champ. Les méthodes pour mettre à jour la valeur d'un champ doivent commencer par set suivi du nom du champ Les méthodes pour créer des objets (factory) devraient commencer par new ou create Les méthodes de conversion devraient commencer par to suivi par le nom de la classe renvoyée à la suite de la conversion	
Les variables	La première lettre est obligatoirement une minuscule et ne devrait pas être un caractère dollar '$' ou underscore '_' même si ceux ci sont autorisés. Pour les variables d'instances non publiques, certains recommandent de commencer par un underscore pour éviter la confusion avec le nom d'une variable fournie en paramètre d'une méthode tel que le setter. Si le nom est composé de plusieurs mots, la première lettre de chaque mot doit être en majuscule, ne pas mettre de caractère underscore '_'. Les noms de variables composés d'un seul caractère doivent être évités sauf pour des variables provisoires (index d'une boucle). Les noms communs pour ces variables provisoires sont i, j, k, m et n pour les entiers et c, d et e pour les caractères.	String nomPersonne; Date dateDeNaissance; int i;
Les constantes	Toujours en majuscules, chaque mots est séparés par un underscore '_'. Ces variables doivent obligatoirement être initialisées lors de leur déclaration.	static final int VAL_MIN = 0; static final int VAL_MAX = 9;

75.4. Les séparateurs

L'usage des séparateurs tels que les retours à la ligne, les lignes blanches, les espaces, etc ... permet de rendre le code moins « dense » et donc plus lisibles.

75.4.1. L'indentation

L'unité d'indentation est constituée de 4 espaces. Il n'est pas recommandé d'utiliser les tabulations pour l'indentation.

Il est préférable d'éviter les lignes contenant plus de 80 caractères.

75.4.2. Les lignes blanches

Elles permettent de définir des sections dans le code pour effectuer des séparations logiques.

Deux lignes blanches devraient toujours séparer deux sections d'un fichier source et les définitions des classes et des interfaces.

Une ligne blanche devrait toujours être utilisée dans les cas suivants :

- avant la déclaration d'une méthode,
- entre les déclarations des variables locales et la première ligne de code,
- avant un commentaire d'une seule ligne,
- avant chaque section logique dans le code d'une méthode.

75.4.3. Les espaces

Un espace vide devrait toujours être utilisé dans les cas suivants :

- entre un mot clé et une parenthèse.

Exemple :

```
while (i < 10)
```

- après chaque virgule dans une liste d'argument
- tous les opérateurs binaires doivent avoir un blanc qui les précèdent et qui les suivent

Exemple :

```
      a = (b + c) * d
```

- chaque expression dans une boucle for doit être séparée par un espace

Exemple :

```
for (int i; i < 10; i++)
```

- les conversions de type explicites (cast) doivent être suivies d'un espace

Exemple :

```
i = ((int) (valeur + 10));
```

Il ne faut pas mettre d'espace entre un nom de méthode et sa parenthèse ouvrante.

Il ne faut pas non plus mettre de blanc avant les opérateurs unaires tel que les opérateurs d'incrément '++' et de décrément '--'.

Exemple :

```
i++;
```

75.4.4. La coupure de lignes

Il arrive parfois qu'une ligne de code soit très longue (supérieure à 80 caractères).

Dans ce cas, il est recommandé de couper cette ligne en une ou plusieurs en respectant quelques règles :

- couper la ligne après une virgule ou avant un opérateur
- aligner le début de la nouvelle ligne au début de l'expression coupée

Exemple :

```
  maMethode(parametre1, parametre2, parametre3,
            parametre4, parametre5);
```

75.5. Les traitements

Même s'il est possible de mettre plusieurs traitements sur une ligne, chaque ligne ne devrait contenir qu'un seul traitement

Exemple :

```
i = getSize();
i++;
```

75.5.1. Les instructions composées

Elles correspondent à des instructions qui utilisent des blocs de code.

Les instructions incluses dans ce bloc sont encadrées par des accolades et doivent être indentées.
L'accolade ouvrante doit se situer à la fin de la ligne qui contient l'instruction composée.
L'accolade fermante doit être sur une ligne séparée au même niveau d'indentation que l'instruction composée.

Un bloc de code doit être définit pour chaque traitement même si le traitement ne contient qu'une seule instruction. Cela facilite l'ajout d'instructions et évite des erreurs de programmation.

75.5.2. L'instruction return

Elle ne devrait pas utiliser de parenthèses sauf si celle ci facilite la compréhension

Exemple :

```
  return;
  return valeur;
  return (isHomme() ? 'M' : 'F');
```

75.5.3. L'instruction if

Elle devrait avoir une des formes suivantes :

Exemple :

```
  if (condition) {
     traitements;
  }

  if (condition) {
     traitements;
  } else {
     traitements;
  }

  if (condition) {
     traitements;
  } else if (condition) {
     traitements;
  } else {
     traitements;
  }
```

Même si cette forme est syntaxiquement correcte, il est préférable de ne pas utiliser l'instruction if sans accolades :

Exemple :

```
    if (i == 10) i = 0;  // cette forme ne doit pas être utilisée
```

75.5.4. L'instruction for

Elle devrait avoir la forme suivante :

Exemple :

```
for ( initialisation; condition; mise à jour) {
     traitements;
  }
```

75.5.5. L'instruction while

Elle devrait avoir la forme suivante :

Exemple :

```
while (condition) {
     traitements;
  }
```

S'il n'y a pas de traitements, la forme est la suivante :

```
while (condition);
```

75.5.6. L'instruction do-while

Elle devrait avoir la forme suivante :

Exemple :

```
do {
   traitements;
} while ( condition);
```

75.5.7. L'instruction switch

Elle devrait avoir la forme suivante :

Exemple :

```
switch (condition) {
case ABC:
   traitements;
case DEF:
   traitements;
case XYZ:
   traitements;
default:
   traitements;
}
```

Il est préférable de terminer les traitements de chaque cas avec une instruction break.

Toutes les instructions switch devrait avoir un cas 'default' en fin d'instruction.

Même si elle est redondante, une instruction break devrait être incluse en fin des traitements du cas 'default'.

75.5.8. Les instructions try-catch

Elle devrait avoir la forme suivante :

Exemple :

```
try {
   traitements;
} catch (Exception1 e1) {
   traitements;
} catch (Exception2 e2) {
   traitements;
} finally {
   traitements;
}
```

75.6. Les règles de programmation

75.6.1. Le respect des règles d'encapsulation

Il ne faut pas déclarer de variables d'instances ou de classes publiques sans raison valable.

Il est préférable de restreindre l'accès à la variable avec un modificateur d'accès protected ou private et de déclarer des méthodes respectant les conventions instaurées dans les javaBeans : getXxx() ou isXxx() pour obtenir la valeur et setXxx() pour mettre à jour la valeur.

La création de méthodes sur des variables private ou protected permet d'assurer une protection lors de l'accès à la variable (déclaration des méthodes d'accès synchronized) et éventuellement un contrôle lors de la mise à jour de la valeur.

75.6.2. Les références aux variables et méthodes de classes.

Il n'est pas recommandé d'utiliser des variables ou des méthodes de classes à partir d'un objet instancié : il ne faut pas utiliser objet.methode() mais classe.methode().

Exemple à ne pas utiliser si afficher() est une méthode de classe :

```
MaClasse maClasse = new MaClasse();
maClasse.afficher();
```

Exemple à utiliser si afficher() est une méthode de classe :

```
MaClasse.afficher();
```

75.6.3. Les constantes

Il est préférable de ne pas utiliser des constantes numériques en dur dans le code mais de déclarer des constantes avec des noms explicites. Une exception concerne les valeurs -1, 0 et 1 dans les boucles for.

75.6.4. L'assignement des variables

Il n'est pas recommandé d'assigner la même valeur à plusieurs variables sur la même ligne :

Exemple :

```
i = j = k;                    //cette forme n'est pas recommandée
```

Il ne faut pas utiliser l'opérateur d'assignement imbriqué.

Exemple à proscrire :

```
  valeur = (i = j + k ) + m;
```

Exemple :

```
  i = j + k;
  valeur = i + m;
```

Il n'est pas recommandé d'utiliser l'opérateur d'assignation = dans une instruction if ou while afin d'éviter toute confusion.

75.6.5. L'usage des parenthèses

Il est préférable d'utiliser les parenthèses lors de l'usage de plusieurs opérateurs pour éviter des problèmes liés à la priorité des opérateurs.

Exemple :

```
if (i == j && m == n)                // à éviter
if ( (i == j) && (m == n ))          // à utiliser
```

75.6.6. La valeur de retour

Il est préférable de minimiser le nombre d'instruction return dans un bloc de code.

Exemple à eviter :

```
if (isValide()){
   return true;
} else {
  return false;
}
```

Exemple :

```
return isValide();
```

Exemple :

```
if (isValide()) {
   return x;
} else return y;
```

Exemple à utiliser :

```
return (isValide() ? x : y)
```

75.6.7. La codification de la condition dans l'opérateur ternaire ? :

Si la condition dans un opérateur ternaire ? : contient un opérateur binaire, cette condition doit être mise entre parenthèses

Exemple :

```
( i >= 0 ) ? i : -i;
```

75.6.8. La déclaration d'un tableau

Java permet de déclarer les tableaux de deux façons :

Exemple :

```
public int[] tableau = new int[10];
public int tableau[] = new int[10];
```

L'usage de la première forme est recommandé.

76. L'encodage des caractères

Un caractère est une unité minimale abstraite de texte qui n'a pas forcement toujours la même représentation graphique.

La plate-forme Java utilise Unicode pour son support des caractères mais il est fréquent de devoir traiter des données textuelles encodées différemment en entrée ou en sortie d'une application. Java propose plusieurs classes et méthodes pour permettre la conversion de nombreux encodages de caractères de et vers Unicode.

Les applications Java qui doivent traiter des données non encodées en Unicode, sont lues avec l'encodage adéquat, stockées et traitées en Unicode et exportent le résultat de Unicode vers l'encodage initial ou l'encodage cible.

La version 5.0 de Java propose un support de la version 4.0 d'Unicode.

La JSR 204 « Unicode Supplementary Character Support » définit le support des caractères étendus d'Unicode dans la plate-forme Java. Ceci permet le support des caractères au-delà des 65546 possibles sur un stockage dans 2 octets.

76.1. L'utilisation des caractères dans la JVM

Une chaîne de caractères est stockée en interne dans la JVM en UTF-16. L'encodage des caractères est uniquement à réaliser en entrée ou en sortie de la JVM (fichiers, base de données, flux, ...).

76.1.1. Le stockage des caractères dans la JVM

En interne, Java utilise le jeu de caractères Unicode et stocke les caractères encodés en UTF-16. Cependant pour le stockage persistant ou l'échange de données, il peut être nécessaire d'utiliser différents jeux d'encodage de caractères. Java propose des mécanismes au travers d'API pour permettre ces conversions.

Le type de données primitif char qui stocke un caractère à une représentation sous la forme d'un entier de 16 bits non signé pour pouvoir contenir un caractère encodé en UTF-16. Toutes les classes qui encapsulent un ou plusieurs caractères encodent ceux-ci en UTF-16 en interne dans la JVM.

76.1.2. L'encodage des caractères par défaut

Pour modifier l'encodage utilisé par défaut pour la lecture et l'écriture dans des flux, il faut modifier la valeur de la propriété file.encoding de la JVM.

Il est possible de fournir la valeur désirée en paramètre de la JVM.

Exemple :

```
java.exe "-Dfile.encoding=UTF-8" -jar monapp.jar
```

La propriété peut aussi être modifiée par programmation en utilisant la méthode setProperty() de la classe System.

Exemple :

```
System.setProperty( "file.encoding", "UTF-8" );
```

76.2. Les jeux d'encodage de caractères

Il existe de nombreux jeux d'encodage de caractères. Une liste complète des jeux d'encodage de caractères est consultable à l'url http://www.iana.org/assignments/character-sets

Un jeu de caractères est un ensemble de caractères. Dans un jeu, chaque caractère est associé à un numéro unique.

Les jeux de caractères les plus utilisés dans les pays occidentaux sont notamment ISO-8859-1, ISO-8859-15, UTF-8, Windows CP-1252, ...

76.3. Unicode

Unicode est un ensemble de caractères pouvant contenir tous les caractères utilisés dans le monde. Unicode peut contenir jusqu'à 1 million de caractères, mais tous les caractères ne sont pas utilisés.

L'ensemble des caractères est divisés en blocs.

Unicode est géré par un consortium : la version courante d'Unicode est la 5.

Unicode attribut à chaque caractère un identifiant nommé code point. Unicode utilise la notation hexadécimale prefixée par « U+ » pour représenter un code point : exemple avec le caractère A qui possède le numéro U+0041.

Les 127 premiers caractères d'Unicode correspondent exactement à l'ensemble de caractères Ascii.

76.3.1. L'encodage des caractères Unicode

Les caractères Unicode peuvent être encodés avec plusieurs encodages de la norme UTF (Unicode Transformation Format)

UTF-32 est l'encodage le plus simple d'Unicode puisqu'il utilise 32 bits (4 octets) pour stocker chaque caractère mais c'est aussi l'encodage le plus coûteux en mémoire.

UTF-16 utilise un encodage sur 16 bits (2 octets) ou 2 fois 16 bits pour stocker les caractères Unicode. Ainsi les valeurs comprises entre U+0000 et U+FFFF sont encodées uniquement sur 16 bits. Les valeurs au-delà sont stockées sur 2 fois 16 bits.
Son principal avantage est qu'il est capable de stocker la plupart des caractères courants avec un seul entier de 16 bits.

Remarque : les fichiers encodés en UTF-16 ne sont généralement pas échangeables entre différents systèmes car deux conventions d'ordonnancement des octets sont utilisées.

UTF-8 utilise un encodage sur 1 à 4 octets pour stocker les caractères Unicode selon leurs valeurs :

- entre U+0000 et U+007F : elles sont stockées sur un seul octet
- entre U+0080 et U+07FF : elles sont stockées sur deux octets
- entre U+0800 et U+FFFF : elles sont stockées sur trois octets
- entre U+10000 to U+10FFFF : elles sont stockées sur quatre octets

Avec UTF-8 chaque caractère est encodé sur un nombre variable d'octets. L'avantage d'UTF-8 est qu'il est compatible avec l'Ascii puisque les premiers caractères sont ceux de la table Ascii et qu'ils sont codés sur un seul octet en UTF-8. Ceci rend UTF-8 assez largement utilisé.

L'encodage/décodage en UTF-8 est assez coûteux car complexe puisque les caractères sont encodés sur un nombre variable d'octets.

UTF-7 encode un caractère Unicode grâce à des séquences de caractères Ascii 7 bits. Cet encodage est utilisé par certains protocoles de messagerie.

Le tableau ci-dessous montre les valeurs des octets de différents encodages de quelques caractères Unicode.

Symbole	A	Z	0	9	€	é	@
Code point	U+0041	U+005A	U+0030	U+0039	U+20AC	U+00E9	U+0040
UTF-8	41	5A	30	39	E2 82 AC	C3 A9 20	40
UTF-16 Litte indian	41 00	5A 00	30 00	39 00	AC 20	E9 00	40 00
UTF-16 Big indian	00 41	00 5A	00 30	00 39	20 AC	00 E9	00 40
UTF-32 Little indian	41 00 00 00	5A 00 00 00	30 00 00 00	39 00 00 00	AC 20 00 00	E9 00 00 00	40 00 00 00
UTF-32 Big indian	00 00 00 41	00 00 00 5A	00 00 00 30	00 00 00 39	00 00 20 AC	00 00 00 E9	00 00 00 40

76.3.2. Le marqueur optionnel BOM

Le début d'un fichier encodé en UTF peut contenir un marqueur optionnel nommé BOM (Byte Order Marker). Ce marqueur a deux utilités :

- Permet de préciser que le texte est encodé en UTF-8, UTF-16 ou UTF-32
- Pour UTF-16- et UTF-32, il permet de préciser l'ordre des octets (little-indian ou big-indian)

En UTF-8, les trois premiers octets du BOM sont EF BB BF.

Lorsque l'on édite un fichier encodé en UTF-8 contenant un BOM avec un éditeur utilisant l'encodage iso-8859-1, les premiers octets affichés sont `ï»¿`

En UTF-16, les deux premiers octets du BOM peuvent avoir deux valeurs :

- FE FF : pour préciser que c'est big-indian est utilisé
- FF FE : pour préciser que c'est little-indian est utilisé

En UTF-32, les quatre premiers octets du BOM peuvent avoir deux valeurs :

- 00 00 FE FF : pour préciser que c'est big-indian est utilisé
- FF FE 00 00 : pour préciser que c'est little-indian est utilisé

76.4. L'encodage de caractères

L'environnement d'exécution Java supporte en standard plusieurs jeux d'encodage de caractères dont :

- US-ASCII : encodage des caractères sur 7 bits
- ISO-8859-1 (latin 1) : encodage des caractères sur 8 bits dans un seul octet pour la plupart des caractères des langues européennes de l'ouest excepté le caractère €
- ISO-8859-15 : comme l'ISO-8859-1 sauf 8 caractères qui sont différents dont le caractère €
- UTF-8 : encodage des caractères sur 8 bits dans 1 à 4 octets. Les caractères ASCII correspondent aux premiers caractères en UTF-8. Peut commencer par un ensemble d'octets optionnels nommés BOM (EF BB BF)
- UTF-16 : encodage des caractères sur 16 bits dans 2 ou 4 octets. Il permet l'encodage de tous les caractères utilisés dans le monde (deux encodages existent : UTF-16 BE et UTF-16 LE)
- Cp1252 : variante utilisée par Microsoft Windows du latin 1
- ...

Les implémentations de l'environnement d'exécution Java proposent généralement un ensemble beaucoup plus complet de jeux d'encodage de caractères. Une liste complète des jeux d'encodage de caractères supportés par la plate-forme Java est consultable à l'url : http://java.sun.com/j2se/1.4.2/docs/guide/intl/encoding.doc.html

Plusieurs classes qui manipulent des caractères permettent de convertir les caractères au format Unicode en utilisant le jeu d'encodage de caractères souhaité notamment :

- java.lang.String
- java.io.InputStreamReader
- java.io.OutputStreamWriter
- et les classes du package java.nio.charset

Les jeux d'encodage de caractères supportés par la plate-forme Java dépendent de leur implémentation. Les jeux de caractères supportés en standard sont stockés dans le fichier rt.jar

Attention : la désignation des jeux de caractères dans les API java.io et java.lang sont différentes de la désignation de ceux dans l'API java.nio. Exemple :

Description	Nom pour l'API java.io et java.lang	Nom pour l'API java.nio<
MS-DOS Latin-2	Cp852	IBM852
ISO Latin 1	ISO8859_1	ISO-8859-1
ISO Latin 2	ISO8859_2	ISO-8859-2
ISO Latin 15	ISO8859_15	ISO-8859-15
ASCII	ASCII	US-ASCII
UTF 8	UTF8	UTF-8
UTF 16	UTF-16	UTF-16
UTF 32	UTF_32	UTF-32
Windows Latin 1	Cp1252	windows-1252

Les jeux de caractères supportés par la version internationale sont dans le fichier charsets.jar du sous répertoire lib du répertoire d'installation du JRE.

Le plus important lorsque l'on manipule des données de type texte en Java est de s'assurer que les caractères seront encodés ou décodés avec le bon jeu de caractères d'encodage.

Une fois que la conversion est faite en écriture et en lecture, le support des caractères Unicode se fera de façon transparente entre l'application Java et les ressources externes.

Par exemple, pour écrire des données de type texte dans un fichier, il suffit de préciser le jeu de caractères d'encodage.

Lors de la lecture de ce fichier, il suffit de préciser le même jeu de caractères d'encodage pour obtenir les données.

Chaque fichier encodé est composé d'un ensemble d'un ou plusieurs octets. Java travaille en interne en stockant les données de types caractères ou chaîne de caractères en Unicode en utilisant l'encodage UTF-16.

L'encodage se fait toujours du type String vers le type byte[].

Le décodage se fait toujours du type byte[] vers le type String.

76.4.1. Les classes du package java.lang

La classe String permet aussi des conversions d'Unicode vers un type d'encodage et vice et versa.

Un constructeur de la classe String permet de créer une chaîne de caractères à partir d'un tableau d'octets et du nom de l'encodage utilisé.

Exemple :

```
byte[] someBytes = ...;
String encodingName = "Shift_JIS";
         String s = new String ( someBytes, encodingName );
```

Pour obtenir un tableau d'octets qui contient le contenu d'une chaîne de caractères encodée selon un encodage particulier, il faut utiliser la méthode getBytes() de la classe String

Exemple :

```
// Using String.getBytes to encode String to bytes
String s = ...;
byte [] b = s.getBytes( "8859_1" /* encoding */ );
```

La méthode getBytes() de la classe String utilise par défaut l'encodage du système d'exploitation sur lequel la JVM est exécutée.

Une surcharge de la méthode getBytes() permet de préciser l'encodage à utiliser.

76.4.2. Les classes du package java.io

Les classes qui héritent des classes Reader et Writer proposent des fonctionnalités pour permettre des opérations de lecture et d'écriture de caractères dans un flux.

La classe InputStreamReader permet de lire des données encodées avec de nombreux types d'encodage pour obtenir des données stockées dans la JVM en Unicode.

Exemple :

```
// Pour lire un fichier en UTF8 :
new InputStreamReader(new FileInputStream("monfichier;txt"), "utf8")
```

La classe OutputStreamReader permet d'écrire des données en Unicode encodées vers de nombreux types d'encodage.

Les classes qui héritent de la classe Reader décodent des octets en String en fonction de l'encodage précisé.

Les classes qui héritent de la classe Writer encodent des String en octets en fonction de l'encodage précisé.

Les classes FileReader et FileWriter permettent de lire et d'écrire des caractères dans un flux.

Exemple :

```
// FileWriter
Writer w = new BufferedWriter(new
OutputStreamWriter(new FileOutputStream(file), "UTF-8"));
// FileReader
Reader r = new BufferedReader(new InputStreamReader(new
FileInputStream(file), "UTF-8"));
```

76.4.3. Le package java.nio

Le package java.nio.charset propose plusieurs classes pour réaliser des conversions de caractères.

La méthode canEncode() de la classe CharsetEncoder permet de vérifier si une chaîne de caractères peut être encodée avec un jeu de caractères d'encodage.

Exemple : vérifier si une chaîne de caractères peut être encodée en latin-1

```
String str = "abcdef"
CharsetEncoder encoder = Charset.forName("iso-8859-1").newEncoder();
boolean ok = encoder.canEncode(str);
str = "1000€";
encoder =
Charset.forName("iso-8859-1").newEncoder();
ok = encoder.canEncode(str);
```

La méthode availableCharsets() de la classe java.nio.Charset permet de connaître la liste des encodages supportés.

La classe java.nio.Charset permet aussi de faire des conversions. Son grand avantage est de ne pas avoir à rechercher la classe correspondant à l'encodage utilisée à chaque appel comme c'est le cas avec les méthodes de la classe String.

Exemple :

```
import java.nio.ByteBuffer;
import java.nio.CharBuffer;
import java.nio.charset.Charset;

byte[] b = ...;
Charset def = Charset.defaultCharset() ;
Charset cs = Charset.forName("Shift_JIS");
ByteBuffer bb = ByteBuffer.wrap( b );
CharBuffer cb = cs.decode( bb );
String s = cb.toString();
```

Le package java.nio.charset.spi propose des classes pour définir ces propres jeux d'encodage de caractères.

76.5. L'encodage du code source

La plupart des fichiers sources sont encodés en ASCII, ISO-8859-1 ou d'autres mais dans tous les cas, ils sont transformés en UTF-16 avant la compilation.

Le code source peut être écrit directement en utilisant un format UTF, par exemple UTF-8. Il suffit alors de préciser au compilateur le jeu de caractères d'encodage utilisés.

Attention : si le code est écrit en UTF-8, il faut s'assurer que l'éditeur n'inclus pas le BOM (Byte Order Mark) au début du

fichier (par exemple, c'est ce que fait l'outil Notepad sous Windows), sinon le compilateur refusera de compiler le code source

Exemple :

```
public class Test {

  public static void
main(String[] args) {
    System.out.println("€");
  }

}
```

Avec l'outil Notepad, il faut enregistrer le fichier au format utf-8 et compiler la classe en précisant que l'encodage est UTF-8.

Exemple :

```
C:\temp>"C:\Program
Files\Java\jdk1.6.0_07\bin\javac" -encoding utf-8 Test.java
Test.java:1: illegal
character: \65279
?public class Test {
^
1
error
```

En fait, Notepad a ajouté les octets du BOM au début du fichier

Résultat :

```
ï"¿public class Test {
...
```

Sans ces octets, le code source se compile parfaitement sous réserve de bien préciser la valeur utf-8 au paramètre -encoding du compilateur javac.

Il est aussi possible d'utiliser l'outil native2ascii fournit par le JDK. Cet outil lit le code source et le convertit en ascii en échappant les caractères non ascii avec leur représentation hexadécimale. Il n'est alors plus nécessaire d'utiliser le paramètre encoding. L'avantage de cette solution est que le code source est lisible sur tous les systèmes puisqu'il est encodé en Ascii.

L'outil native2ascii, dont le nom est relativement inadéquat, fourni avec le jdk permet de convertir un fichier source d'un encodage vers un encodage Ascii dans lequel tous les caractères non ascii sont échappés sous la forme \unnnn ou nnnn représente le code Unicode du caractère.

76.6. L'encodage de caractères avec différentes technologies

L'encodage de caractères est généralement nécessaire et cela avec plusieurs technologies utilisées en Java.

76.6.1. L'encodage de caractères dans les fichiers

Généralement, les fichiers texte ne contiennent aucunes indications sur l'encodage utilisé.

Certaines normes proposent cependant des fonctionnalités optionnelles pour fournir l'information.

Exemple : HTML

```
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
```

Exemple : XML

```
<?xml version="1.0" encoding="ISO8859-1" ?>
```

76.6.2. L'encodage de caractères dans une application web

Pour utiliser l'encodage UTF-8 dans une application web, il faut prendre plusieurs précautions.

Dans les JSP, il faut définir l'encodage utilisé

Exemple :

```
<%@ page contentType="text/html; charset=UTF-8"%>
<%@ page pageEncoding="UTF-8"%>
```

Dans une servlet, il est possible d'utiliser la méthode setCharacterEncoding() de la classe HttpRequest pour préciser l'encodage des données de la requête. Cet appel doit être fait avant l'utilisation de la méthode getParameter() pour que les données soit correctement décodées.

76.6.3. L'encodage de caractères avec JDBC

Avec JDBC, il est parfois nécessaire de préciser l'encodage utilisé dans les données échangées. Dans ce cas, l'attribut à utiliser dépend de la base de données concernée et il faut consulter la documentation du pilote JDBC utilisé.

Exemple :

```
jdbc:mysql://localhost/mabase?useUnicode=true&characterEncoding=utf8
```

77. Les frameworks

Le développement en Java impose certaines contraintes :

- nécessité de maitriser de nombreux concepts généraux (notamment celui de la POO, les design patterns, etc ...) et spécifiques aux plateformes Java (standard, entreprise et mobile) selon les besoins
- nécessité de maitriser de nombreuses API qui sont de bas niveau
- quasi absence d'outils de type RAD

Ces facteurs allongent la durée de la courbe d'apprentissage des développeurs et complexifient l'architecture des applications.

De ce fait, le concept de framework est apparu, essentiellement massivement soutenu par la communauté open-source qui est très prolifique dans le monde Java.

77.1. La présentation des concepts

Le développement d'applications peut être facilité grâce à l'utilisation :

- de la POO et des design patterns lors de la conception
- de boîtes à outils : ce sont des classes qui proposent des utilitaires basiques
- de frameworks

77.1.1. La définition d'un framework

Le terme framework est fréquemment utilisé dans des contextes différents mais il peut être traduit par cadre de développement.

Les frameworks se présentent sous diverses formes, qui peuvent inclure tout ou partie des éléments suivants :

- un ensemble de classes généralement regroupées sous la forme de bibliothèques pour proposer des services plus ou moins sophistiqués
- un cadre de conception reposant sur les design patterns pour proposer tout ou partie d'un squelette d'applications
- des recommandations sur la mise en oeuvre et des exemples d'utilisation
- des normes de développement
- des outils facilitant la mise en oeuvre

L'objectif d'un framework est de faciliter la mise en ouvre des fonctionnalités de son domaine d'activité. Il doit permette au développeur de se concentrer sur les tâches spécifiques à l'application à développer plutôt qu'à des tâches techniques récurrentes dans le développement d'applications tel que :

* l'architecture de base de l'application
* l'accès aux données
* l'internationalisation
* la journalisation des événements (logging)
* la sécurité (authentification et gestion des rôles)
* le paramétrage de l'application
* ...

La mise en oeuvre d'un framework permet notamment :

* de capitaliser le savoir-faire sans "réinventer la roue"
* d'accroître la productivité des développeurs une fois le framework pris en main
* d'homogénéiser les développements des applications en assurant la réutilisation de composants fiables
* donc de faciliter la maintenance notamment évolutive des applications

Cependant, cette mise en oeuvre peut se heurter à certaines difficultés :

* le temps de prise en main du framework par les développeurs peut être plus ou long en fonction de différents facteurs (complexité du framework, richesse de sa documentation, expérience des développeurs, ...)
* les évolutions du framework qu'il faut répercuter dans les applications existantes

77.1.2. L'utilité de mettre en oeuvre des frameworks

Le développement d'applications d'entreprise avec Java (J2EE) s'avère relativement complexe. Il est nécessaire d'assimiler de nombreux concepts et API pour pouvoir développer et déployer une application J2EE. Par exemple pour le développement d'applications web, les API et les spécifications de J2EE ne proposent qu'un support de bas niveau au travers des API Servlets et JSP.

Le développement d'une application de taille moyenne ou complexe va ainsi rencontrer d'énormes difficultés. Pour faciliter le développement et ainsi augmenter la productivité, des frameworks ont été développés.

L'intérêt de la mise en oeuvre d'un ou plusieurs frameworks a fait ses preuves depuis longtemps. Une des grandes difficultés est de sélectionner le ou les frameworks à utiliser.

L'intérêt majeur des frameworks est de proposer une structure identique pour toutes les applications qui l'utilisent et de fournir des mécanismes plus ou moins sophistiqués pour assurer des tâches communes à toutes les applications.

Il est possible de développer son propre framework mais cela représente un investissement long, risqué et donc coûteux qu'il sera quasi impossible de rentabiliser d'autant que de nombreux frameworks sont présents sur le marché dont quelques frameworks open source particulièrement matures. Par exemple, le framework Struts du groupe Apache Jakarta est devenu un quasi standard adopté par la majorité des acteurs proposant un IDE et sert de base au développement d'autres frameworks open source ou commerciaux.

Il existe de nombreux frameworks open source dont la possibilité de mise en oeuvre concrète est très variable en fonction de plusieurs facteurs : fonctionnalités proposées, maturité du projet, évolutions constantes, documentation proposée, ...

Plusieurs d'entre eux sont devenus de véritable succès grâce à leur adoption par de nombreux développeurs et à l'apport de nombreux contributeurs (exemple : Struts, Log4J ou Spring). Ils permettent d'avoir des frameworks relativement complets, fiables et fonctionnels. En plus, comme tout projet open source, les sources sont disponibles ce qui permet éventuellement de faire des modifications pour répondre à ces propres besoins ou ajouter des fonctionnalités.

La diversité de ces frameworks permet de répondre à de nombreux besoins. Le revers de la médaille est la difficulté de choisir celui ou ceux qui répondront au besoin.

Pour choisir un framework, les caractéristiques suivantes doivent être prises en compte :

* adoption par la communauté
* la qualité de la documentation
* le support (commercial ou communautaire)
* le support par les outils de développement

Le plus gros défaut des frameworks est lié à leur complexité : il faut un certain temps d'apprentissage pour avoir un minimum de maitrise et d'efficacité dans leur utilisation.

Le choix d'un framework est très important car l'utilisation d'un autre framework en remplacement impose souvent un travail important essentiellement lié à la prise en main du nouveau framework et aux adaptions ou la réécriture partielle de morceaux de l'application.

77.1.3. Les différentes catégories de framework

Généralement, le coeur d'une application repose sur une architecture proposée par un framework mais il est aussi nécessaire de prévoir d'autres frameworks pour réaliser certaines tâches généralement techniques :

- logging
- mapping O/R
- automatisation des tests
- ...

Ainsi, les frameworks peuvent être regroupés en plusieurs catégories :

- Technique : propose des services techniques récurrents
- Structurel : propose la mise en place d'une architecture applicative
- Métier : propose des services fonctionnels
- Tests : propose des services pour automatiser les tests unitaires

77.1.4. Les socles techniques

Généralement, pour le développement d'applications, il est nécessaire de mettre en place une architecture et d'utiliser plusieurs frameworks dédiés à la mise en oeuvre des fonctionnalités auxquelles ils répondent. Cet ensemble d'entités est désigné par socle technique.

Un socle technique est donc composé d'une architecture et d'un ensemble de frameworks pour faciliter le développement des couches qui composent l'architecture (IHM, objets métiers, persistance des données, ...) et pour proposer des services techniques transverses (logging, ...)

Les attentes dans la définition d'un socle technique sont nombreuses :

- sélection des frameworks à mettre en oeuvre
- sélection des outils à utiliser
- définition de l'architecture applicative et de ses couches
- création d'un projet "vide" par assemblage des frameworks et des classes de bases des différentes couches
- validation par un prototype
- définition de normes de développement
- apprentissage du socle (plusieurs semaines de pratique sont nécessaires pour être généralement pleinement opérationnel)
- rédaction de la documentation technique
- ...

Généralement, un socle technique doit être défini spécifiquement pour les besoins de l'application ou des applications. Cependant, il est possible de s'appuyer sur un socle existant pour ne pas avoir à complètement la roue. Il existe des socles techniques open source comme par exemple le socle technique scub foundation dont le site officiel est à l'url http://www.scub-foundation.org/.

77.2. Les frameworks pour les applications web

Les applications web sont généralement le point d'entrée principal pour les applications développées en utilisant la plate-forme J2EE.

Pourtant le développement d'applications web est relativement compliqué à concevoir et à implémenter pour plusieurs raisons :

- la nature du protocole http (basé sur un modèle simple de request/response sans état)
- les nombreuses technologies à mettre en oeuvre (HTML, XHTML, CSS, Javascript, ...) ainsi que leurs différentes versions
- le support de ces technologies par les différents navigateurs est particulièrement différent
- HTML est un langage pauvre qui ne permet que le développement de formulaires assez rudimentaires

A ces inconvénients indépendants de la technologie côté serveur pour les mettre en oeuvre viennent s'ajouter des raisons spécifiques à la plate-forme J2EE :

- les API Servlet et JSP sont de bas niveau
- le manque de modèles événementiels (cet argument explique le développement de la technologie JSF)

Dès que l'on développe des applications web uniquement avec les API servlet et JSP, il apparait évident que de nombreuses parties dans des applications différentes sont communes mais doivent être réécrites ou réutilisées à chaque fois. Ceci est en grande partie liée au fait que les API servlets et JSP sont des API de bas niveau. Elles ne proposent par exemple rien pour automatiser l'extraction des données de la requête HTTP et mapper leur contenu dans un objet de façon fiable, assurer les transitions entre les pages selon les circonstances, ...

Les frameworks de développement web permettent généralement une séparation logique d'une application selon le concept proposé par le modèle MVC (Modèle/Vue/Contrôleur). Le framework le plus utilisé dans cette catégorie est Struts.

Certains éditeurs proposent des frameworks qui s'appui sur un framework open source et facilitent leur utilisation en proposant des fonctionnalités dédiés dans leur IDE (Oracle ADF avec Jdeveloper, Beehive avec Weblogic Workshop, ...).

L'utilisation d'un framework web permet de faciliter le développement et la maintenance évolutive d'une application web. Les frameworks utilisent ou peuvent être complétés par des moteurs de templates qui facilitent la génération de page web à partir de modèles.

Les frameworks les plus récents (tel que Java Server Faces) mettent en oeuvre l'utilisation de composants côté serveur pour faciliter les développements (modèle événementiel et développement graphique)

Récemment, une nouvelle forme d'applications web est apparue : les applications riches. Elles peuvent prendre plusieurs formes et notamment dans les développements J2EE utiliser AJAX (reposant sur DHML pour des échanges asynchrones avec le serveur) ou Lazlo (reposant sur flash). Ces technologies rendent les applications plus riches et plus conviviales pour les rapprocher de ce que les utilisateurs connaissent avec le client lourd.

77.3. L'architecture pour les applications web

77.3.1. Le modèle MVC

Le modèle MVC (Model View Controler) a été initialement développé pour le langage Smalltalk dans le but de mieux structurer une application avec une interface graphique.

Ce modèle est un concept d'architecture qui propose une séparation en trois entités des données, des traitements et de l'interface :

- le Modèle représente les données de l'application généralement stockées dans une base de données

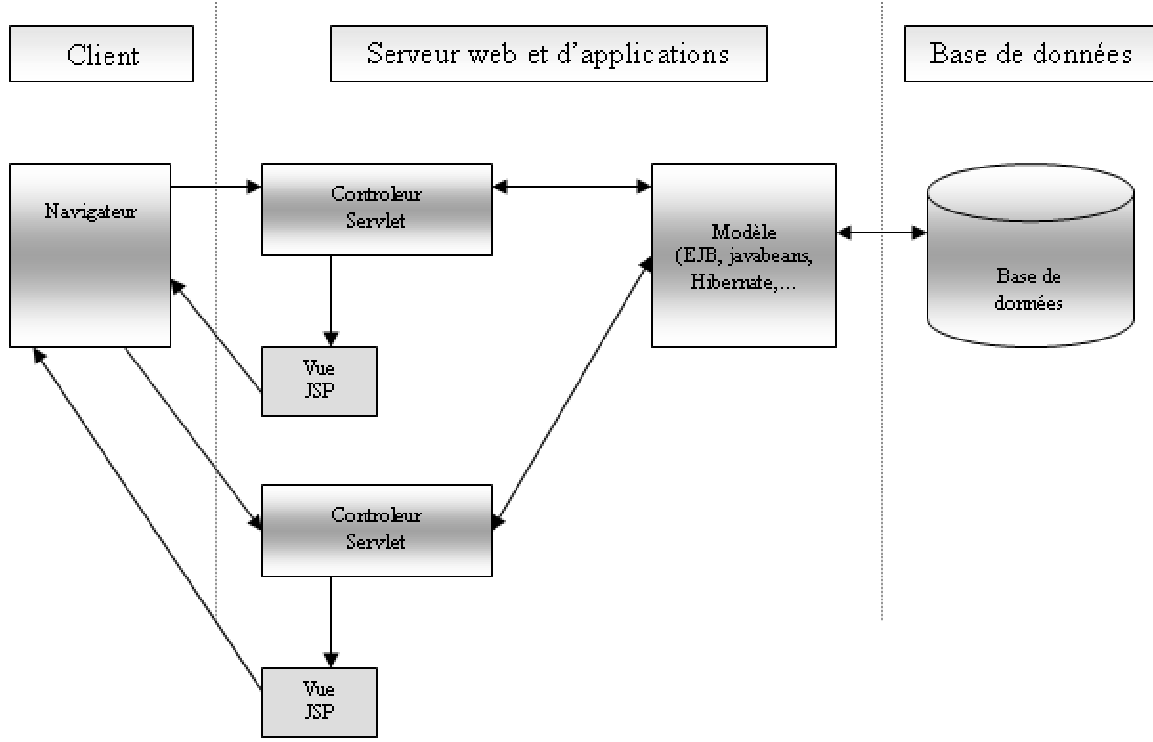
* la Vue correspond à l'IHM (Interface Homme Machine)
* le Contrôleur assure les échanges entre la vue et le modèle notamment grâce à des composants métiers

Initialement utilisé pour le développement des interfaces graphiques, ce modèle peut se transposer pour les applications web sous la forme d'une architecture dite 3-tiers : la vue est mise en oeuvre par des JSP, le contrôleur est mis en oeuvre par des servlets et des Javabeans. Différents mécanismes peuvent être utilisés pour accéder aux données.

L'utilisation du modèle MVC rend un peu plus compliqué le développement de l'application qui le met en oeuvre mais il permet une meilleure structuration de l'application.

77.4. Le modèle MVC type 1

Dans ce modèle, chaque requête est traitée par un contrôleur sous la forme d'une servlet. Celle-ci traite la requête, fait appel aux éléments du model si nécessaire et redirige la requête vers une JSP qui se charge de créer la réponse à l'utilisateur.



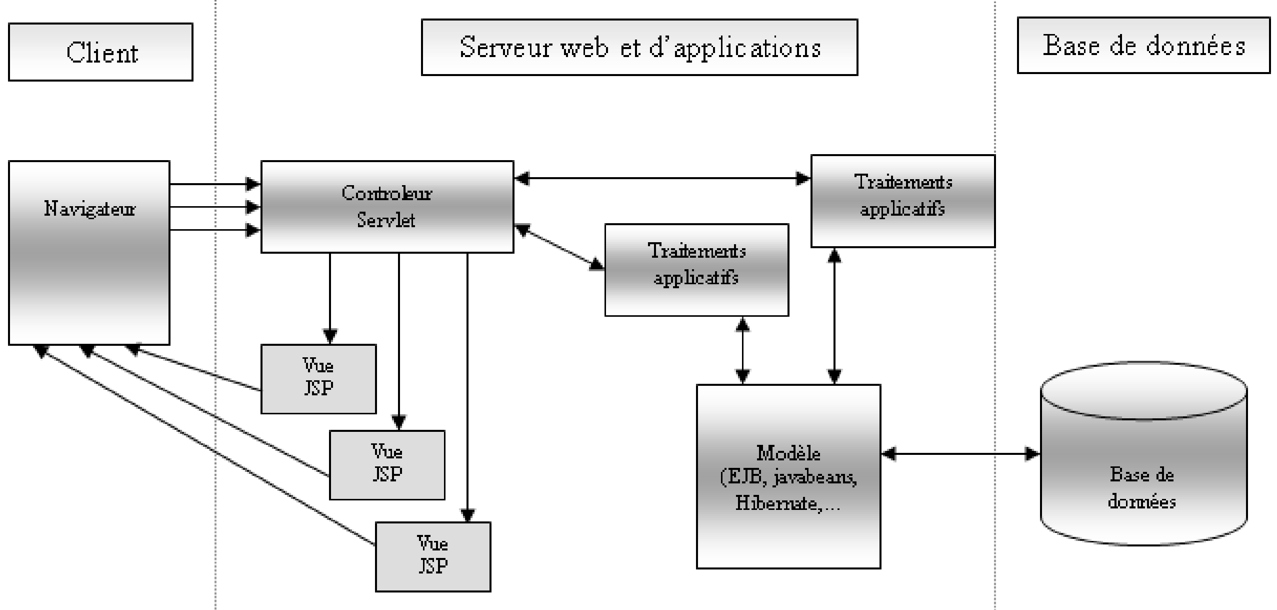
L'inconvénient est donc une multiplication du nombre de servlets nécessaire à l'application : l'implémentation de plusieurs servlets nécessite beaucoup de code à produire d'autant que chaque servlet doit être déclarée dans le fichier web.xml.

77.5. Le modèle MVC de type 2

Le principal défaut du modèle MVC est le nombre de servlets à développer pour une application.

Pour simplifier les choses, le modèle MVC model 2 ou MVC2 de Sun propose de n'utiliser qu'une seule et unique servlet comme contrôleur. Cette servlet se charge d'assurer le worflow des traitements en fonction des requêtes http reçues.

Le modèle MVC 2 est donc une évolution du modèle 1 : une unique servlet fait office de contrôleur et gère toutes les requêtes à traiter en fonction d'un paramétrage généralement sous la forme d'un fichier au format XML.



Le modèle MVC II conserve les principes du modèle MVC, mais il divise le contrôleur en deux parties en imposant un point d'entrée unique à toute l'application (première partie du contrôleur) qui déterminera à chaque requête reçue les traitements applicatifs à invoquer dynamiquement (seconde partie du contrôleur).

Une application web implémentant le modèle MVC de type II utilise une servlet comme contrôleur traitant les requêtes. En fonction de celles-ci, elle appelle les traitements dédiés généralement encapsulés dans une classe.

Dans ce modèle, le cycle de vie d'une requête est le suivant :

1. Le client envoie une requête à l'application qui est prise en charge par la servlet faisant office de contrôleur.
2. La servlet analyse la requête et appelle la classe dédiée contenant les traitements
3. Cette classe exécute les traitements nécessaires en fonction de la requête notamment en faisant appel aux objets métiers.
4. En fonction du résultat des traitements, la servlet redirige la requête vers la page JSP
5. La JSP génère la réponse qui est renvoyée au client

77.5.1. Les différents types de framework web

Il existe deux grandes familles de framework web pour les applications J2EE : ceux reposant sur les requêtes et ceux reposant sur les composants.

Cette dernière famille est assez récente : elle repose aussi sur HTTP (donc sur les requêtes) mais propose de faciliter le développement et surtout de rendre plus riche les applications en utilisant des composants dont le rendu est généré par le serveur avec une gestion des événements. Ce type de framework va devenir la référence dans un future proche et sont proposés comme la nouvelle génération de framework pour le développement web.

En mars 2002, Tapestry a été le premier framework de ce type en open source diffusé. En Mars 2004, le JCP propose les Java Server Faces qui sont le standard de framework web intégré dans Java EE 5.

Les frameworks reposant sur le traitement de requêtes utilisent un cadre de développement utilisant les servlets et les JSP pour faciliter le traitement de certaines tâches répétitives (extraction des données de la requête, enchainement des pages, utilisation de bibliothèques de tags spécifiques pour l'IHM, ...). Ces framework reposent généralement sur le modèle MVC2.

Exemple : Struts, WebWork, Spring MVC, ...

77.5.2. Des frameworks pour le développement web

Les frameworks présentés dans cette section regroupent les principaux frameworks open source et les JSF. Les framework open source sont nombreux : Anvil, Chrysalis, Echo 2, Echo, Expresso Framework, Jaffa, Jucas, Maverik, Millstone, SOFIA, Spring MVC, Struts , Tapestry, Turbine, VRaptor, WebOnSwing, WebWork, Wicket, ...

Seuls les principaux sont rapidement présentés dans les sections suivantes. Certains font l'objet d'un chapitre détaillé dédié.

77.5.2.1. Apache Struts

Struts est un projet du groupe Jakarta de la fondation Apache. La page officielle de Struts est à l'url : http://struts.apache.org/.

Struts est le framework open source le plus populaire : son utilisation est largement répandue.

Son architecture met en oeuvre le modèle MVC 2. Il utilise les servlets, les JSP, XML, les resourcesbundles (pour l'internationnalisation) et des bibliothèques du projet Jakarta Commons.

Struts se concentre sur le contrôleur et la vue. Il ne propose rien concernant le modèle ce qui laisse le développeur libre pour mettre en oeuvre des Javabeans, des EJB ou tout autre solution pour la persistance des données.

La partie vue peut utiliser les tags proposés par Struts qui sont assez fournis mais qui nécessitent souvent l'ajout de bibliothèques supplémentaires pour gagner en productivité (exemple : Struts-Layout). Il est aussi possible d'utiliser ces propres tags et/où d'utiliser la JSTL (Java Standard Tag Libraries).

Struts possède cependant quelques faiblesses : de conception assez anciennes, il est techniquement inférieure à d'autres framework notamment ceux reposant sur les composants.
Il est aussi légitime de s'interroger sur le futur de Struts notamment avec l'arrivée des JSF qui vont devenir le standard de la plate-forme J2EE et que le créateur de Struts participe activement aux spécifications des JSF par le JCP.

Les atouts de Struts sont :

- une bibliothèque de tags mature, robuste et complète.
- l'introspection des objets relatifs aux formulaires
- les formulaires de type DynaActionForm
- la validation est extensible, elle peut être faite côté client ou côté serveur.
- la gestion centralisée des exceptions
- la décomposition de l'application en modules logiques reposant sur le modèle MVC 2
- le support de l'internationalisation

Avantages	Inconvénients
Est quasiment un standard de fait	Son avenir
Nombreuses sources d'informations	Peu d'évolutions
Bibliothèques de tags HTML	une architecture vieillissante
Intégration dans les IDE	beaucoup de classe et de code à produire (partiellement automatisable avec des outils dédiés)
open source	

Struts est détaillé dans le chapitre «Struts».

77.5.2.2. Spring MVC

Le framework web Spring MVC fait partie intégrante du framework Spring. Ce dernier propose une architecture complète pour le développement d'applications J2EE dont le module MVC assure le développement de la partie IHM pour les applications web. L'utilisation de ce module peut être remplacée par d'autre framework notamment Struts grâce à la facilité de Spring de s'interfacer avec d'autres framework.

Le succès de Spring repose principalement sur la facilité qu'il propose à développer des applications J2EE sans EJB tout en proposant les mêmes fonctionnalités que ces derniers grâce notamment à l'utilisation du design pattern IoC et la programmation orienté aspect.

Le page officielle de ce projet est à l'url : www.springframework.org

77.5.2.3. Webwork

Webwork est diffusé en open source depuis mars 2002.

Webwork ce veut simple et puissant : son architecture repose sur les patterns de conception command et Inversion of Control. Il propose de nombreuses fonctionnalités notamment la définition et l'utilisation de templates, les thèmes, la validation des données, ...

La page officielle de ce projet est à l'url : http://opensymphony.com/webwork.

Ce framework est en cours de fusion avec la prochaine version de Struts.

77.5.2.4. Tapestry

Tapestry est un projet open source développé par la fondation Apache.

La page officielle de ce projet est à l'url : http://jakarta.apache.org/tapestry.

L'intérêt pour le projet s'est accru depuis la diffusion des JSF car ils utilisent tous les deux les composants serveur.

La partie vue n'utilise pas de JSP mais des fichiers HTML enrichis avec des tags particuliers.

77.5.2.5. Java Server Faces

Les JSF (Java Server Faces) constituent un framework de composants graphiques hébergés côté serveur utiles pour le développement d'applications Web. C'est un framework basé sur des spécifications du JCP. Il dispose à ce titre d'une implémentation de référence.

Les Java Server Faces sont développés par le JCP sous la JSR 127 pour la version 1.0 et 252 pour la version 1.2.

La page officielle du projet est à l'url : http://java.sun.com/j2ee/javaserverfaces

Les JSF sont construits sur un modèle de développement orienté événementiel à l'image de SWING. Il se compose d'un ensemble d'API servant, notamment,

- à représenter les composants,
- à gérer les états et les événements
- à proposer des dispositifs de validation

Les valeurs ajoutées par ce framework sont :

- spécifications standardisées
- architecture de gestion des états des composants et des données correspondantes.

* extensible pour créer de nouveaux composants
* support pour les différents types de clients (seul le renderer HTML est proposé dans l'implémentation de référence).
* séparation entre le comportement et la présentation : les applications Web basées sur la technologie JSP permettent cette séparation mais elle reste très partielle. Une page JSP ne peut pas mapper plusieurs requêtes http sur un composant graphique ou de gérer un élément graphique en tant qu'objet sans état côte serveur.

Ces spécifications possèdent une implémentation de référence (RI) ainsi que des implémentations open source (MyFaces, ...) et commerciales (ADF Faces, ...).

Avantages	Inconvénients
Standard du JCP	Maturité
Utilisation de composants et de la gestion d'événements	Manque de composants évolués (à développer ou à obtenir d'un tiers)
Possibilité de créer ses propres composants	Développement très orienté sur la vue (celle-ci contient la définition des validations et des méthodes à appeler pour les traitements)
Intégration forte dans certains IDE (Sun Studio Creator, ...) pour permettre des développements de type RAD	Peu de retour sur les performances (cycle de traitement des requêtes avec de nombreuses étapes)
Possibilité d'utiliser plusieurs types de rendu via des toolkits (seul HTML est proposé en standard dans l'implémentation de référence)	
De base, pas besoin d'hériter de classes ou implémenter d'interfaces dédiées : de simples beans suffisent	

Les JSF sont détaillées dans le chapitre «JSF (Java Server Faces)».

77.5.2.6. Struts 2

La version du framework Struts sera constitué d'une fusion des framework Struts et WebWork.

Strut 2 est en cours de développement : une première version devrait être livré au troisième trimestre 2006.

77.5.2.7. Struts Shale

Le projet Struts Shale repose sur les JSF pour proposer de faciliter le développement d'applications web.

Ce projet est incompatible avec le framework Struts 1.x.

La page officielle du projet est à l'url : http://struts.apache.org/shale

77.5.2.8. Expresso

Expresso est un framework open source développé par la société JCorporate.

C'est un framework pour le développement d'applications Web J2EE qui agrège de nombreux autres frameworks chacun dédié à une tâche particulière.

Il repose sur Struts depuis sa version 4.0.

Une des particularités de ce framework est de proposer un ensemble de fonctionnalités assez complètes :

* outils de mapping objet-relationnel pour la persistance des données
* gestion d'un pool de connexions aux bases de données
* workflow
* identification et authentification pour la sécurité
* ...

La version 5.0 de ce framework est disponible depuis octobre 2002. La version 5.5, publié en mai 2004 repose sur Struts 1.1. La version 5.6 est publiée en janvier 2005.

La page officielle de ce projet est à l'url : http://www.jcorporate.com/

77.5.2.9. Jena

La page officielle de ce projet est à l'url : http://jena.sourceforge.net/

77.5.2.10. Echo 2

La page officielle de ce projet est à l'url : http://www.nextapp.com/platform/echo2/echo/

77.5.2.11. Barracuda

Le site officiel de Barracuda est à l'url : http://barracudamvc.org/Barracuda/index.html

77.5.2.12. Stripes

Le but principal de ce framework est d'être simple et facile à mettre en oeuvre. Pour y arriver, Stripes utilise des technologies récentes de Java 5 notamment les annotations pour faciliter la configuration.

La page officiel du projet est à l'url http://mc4j.org/confluence/display/stripes/Home

77.5.2.13. Turbine

La page officiel du projet est à l'url http://jakarta.apache.org/turbine/

77.6. Les frameworks de mapping Objet/Relationel

Les frameworks de mapping Objet/relationel sont détaillés dans le chapitre «La persistance des objets».

77.7. Les frameworks de logging

Le logging est important dans toutes les applications pour permettre de faciliter le débogage lors du développement et de conserver une trace de son exécution lors de l'exploitation en production.

Une API très répandue est celle développée par le projet open source Log4j du groupe Jakarta.

Concient de l'importance du logging, Sun a développé et intégré au JDK 1.4 une API dédiée. Cette API est plus simple à utiliser et elle offre moins de fonctionnalités que Log4j mais elle a l'avantage d'être fournie directement avec le JDK.

Face au dilemme du choix de l'utilisation de ces deux API, le groupe Jakarta a développé une API qui permet d'utiliser indifféremment l'une ou l'autre de ces deux API selon leur disponibilité.

L'utilisation de ces entités est détaillée dans le chapitre «Logging».

Il existe aussi d'autres frameworks de logging dont l'utilisation est largement moins répandues.

78. Les frameworks de tests

Le but d'un test est de vérifier qu'une fonctionnalité fait ce qu'on attend d'elle.

Les tests d'une application sont une phase très importante dans les cycles de développement et de maintenance d'une application. Ils permettent de détecter des bugs et de s'assurer que l'application réponde au cahier des charges et aux spécifications.

Ces tests peuvent prendre différentes formes :

- tests unitaires : les tests unitaires automatisés sont un des mécanismes les plus importants pour améliorer la qualité et tenter de garantir la fiabilité du code d'une application.
- tests d'intégration
- tests de recette : le but est de vérifier que l'application réponde aux spécifications fonctionnelles. Ces tests sont faits par les utilisateurs qui devraient fournir un PV de recette
- tests de charge (robustesse, performance, montée en charge, ...)
- tests de stress
- tests d'acceptabilité
- tests de sécurité
- ...

La mise en oeuvre des tests peut être facilitée par l'utilisation d'outils :

- frameworks d'automatisation des tests unitaires : xUnit, TestNG, ...
- outils de code coverage : emma, ...
- outils pour automatiser les tests des IHM : Selenium pour les applications web, ...
- outils pour les tests fonctionnels : Fitnesse, ...
- ...

Ce chapitre va essentiellement se concentrer sur la mise en oeuvre de certains de ces tests avec certains outils.

Ce chapitre contient plusieurs sections :

- ♦ Les tests unitaires
- ♦ Les frameworks et outils de tests

78.1. Les tests unitaires

Les tests unitaires peuvent être réalisés de différentes manières :

- manuelle : par exemple en utilisant les capacités de l'IDE notamment celles du débogueur
- manuelle et reproductible : par exemple en créant pour chaque classe une méthode main qui permet d'exécuter des tests. Ce type de tests nécessite un lancement à la main et une analyse humaine des résultats
- automatisée avec un framework de tests

L'utilisation d'un débogueur peut être pratique pour tester du code fraichement écrit et comprendre son fonctionnement mais il ne permet pas d'automatiser ces tests. En effet, cette technique requiert une intervention manuelle et une

interprétation humaine des résultats.

C'est la même problématique avec l'utilisation de traces via des System.out ou l'écriture dans un fichier : les données de ces traces doivent être analysées par une personne.

L'utilisation de frameworks dédiés à l'automatisation des tests unitaires permet d'assurer une meilleure qualité et fiabilité du code. Cette automatisation facilite aussi le passage de tests de non régression notamment lors des mises à jour du code. De plus, l'utilisation de ces frameworks ne nécessite aucune modification dans le code à tester ce qui sépare clairement les traitements représentés dans le code de leurs tests. Enfin, l'analyse des résultats peut être automatisée puisque chaque résultat de tests possède un statut généralement ok ou en erreur.

Ces frameworks ne sont que des outils qui permettent la mise en oeuvre de tests unitaires mais ils ne dispensent pas d'utiliser une méthodologie pour mettre en oeuvre les tests unitaires.

Un test unitaire se déroule en quatre étapes :

- setup : initialiser des objets ou des ressources
- call : exécuter le code à vérifer
- verify : vérifier des données issuées des traitements
- teardown : permettre de faire le ménage ou de libérer des ressources

Les tests unitaires automatisés sont très importants et ce durant tous le cycle de vie d'une application :

- conception : rédaction de liste des cas de tests
- développement : tests du code, détection précoce de bugs,
- maintenance: tests de non régression, encourage et facilite le refactoring

78.1.1. L'utilité des tests unitaires automatisés

L'utilité des tests unitaires automatisés n'est plus à démontrer : ils sont même primordiaux dans certaines méthodologies notamment XP (eXtreme Programming) et TDD (Test Driven Development). Ils servent à promouvoir et vérifier la qualité et la fiabilité du code.

Les tests unitaires automatisés sont un des outils les plus puissants pour améliorer la qualité d'une application. De plus, l'utilisation de tests unitaires améliore l'organisation et la stabilité du code.

Les tests unitaires n'ont pas qu'un effet de test immédiat du code mais surtout ils permettent d'effectuer des tests de non régression lors de modifications qui interviennent inévitablement durant la vie d'une application. Les tests unitaires automatisés sont donc particulièrement intéressants pour les tests de non régression qui seront automatisés. Il est courant d'avoir des portions de code fréquemment perçues comme mystique car personne ne comprend plus comment il fonctionne malgré le fait que ce code soit primordial. Il est alors toujours délicat de faire évoluer ce code lors de maintenances correctrices ou évolutives.

La présence de tests unitaires automatisés va rassurer le développeur car il pourra réexécuter ces tests avant et après les modifications pour s'assurer qu'il n'y a pas de regression. Bien sûre, le dégré d'assurance augmente avec la complétude et la qualité des tests.

L'écriture de cas de tests permet de prouver que le code à tester fonctionne. Les cas de tests permettent ensuite de s'assurer de la non régression lors des maintenances dans le code. Les tests unitaires permettent de capitaliser sur les tests à effectuer et ainsi de limiter les effets de bord liés aux inévitables modifications correctrices ou évolutives du code.

La POO implique naturellement des dépendances entre les classes. Une modification dans une de ces classes peut facilement induire des effets de bords dans les classes appelantes. Si les tests sont complets et correctes, une modification ayant un effet de bord fera échouer les tests existants. Dans ce cas, soit la modification nécessite une adaptation du cas de tests soit un bug a introduit un effet de bord dans le comportement du code.

L'existence de tests unitaires couvrant une majorité des cas de tests permet d'être plus confiant lors de la modification de code : cela peut améliorer la garantie qu'une modification n'a pas d'effet de bord.

Si une classe possède un ensemble complet de tests unitaires, il y a moins de réticences à faire des modifications dans son code liées à des maintenances correctives, évolutives, pour améliorer les performances ou pour faire du refactoring. Les tests permettent de s'assurer de la non régression des fonctionnalités proposées.

Les risques augmentent dans une application qui ne possède pas ou peux de tests unitaires au fur et à mesure que des modifications sont faites dans le temps. L'absence de tests automatisés implique des tests manuels qui peuvent être oubliés ou mal interprétés augmentant ainsi le risque de ne pas détecter d'effets de bord.

Le coût d'écriture des tests est largement compensé par celui gagné par la réutilisation des tests à chaque itération corrective.

Il est plus facile d'effectuer des opérations de refactoring si les classes disposent d'un ensemble des tests unitaires complets.

Les tests unitaires sont les premiers tests réalisés parmi l'ensemble des tests qui seront réalisés sur l'application. Il ne faut surtout pas les sous estimer en se disant que les tests suivants permettront de détecter les bugs car leur grand avantage est qu'avec un framework dédié ils peuvent être automatisés.

La rédaction de tests unitaires implique nécessairement une amélioration de la conception du code. Il est très facile d'écrire du code lorsque celui ci ne doit pas être testé. Cependant pour écrire du code qui doit être testé, il faut que la conception du code soit adaptée pour faciliter la mise en oeuvre des tests unitaires :

- Améliorer la granularité des méthodes : il est plus facile des tester des méthodes courtes que de longues méthodes
- Réduire la dépendance entre les objets : il est intéressant de mettre en oeuvre certains design patterns afin de réduire le couplage entre les objets
- Une classe avec un couplage fort vers d'autres classes est difficile à tester.

Les tests unitaires peuvent facilement servir d'exemples d'utilisation du code testé puisque le code est nécessairement invoqué durant les tests.

Il est encore fréquent de voir des scénarios de tests écrits dans un document et exécutés manuellement par un humain. Cette approche est obsolète dans la mesure où des outils existent pour automatiser une bonne partie de ces tests évitant ainsi les erreurs humaines (aucune exécution des tests, oublie de l'exécution de cas, mauvaise interprétations des résultats, ...). De plus, les fonctionnalités d'une application ont tendance à augmenter avec le temps ce qui rend ce processus encore plus long et fastidieux.

78.1.2. Quelques principes pour mettre en oeuvre de tests unitaires

Il existe plusieurs approches pour mettre en oeuvre des tests unitaires automatisés : chacune a des avantages et des inconvénients dont il faut tenir compte selon son contexte.

Il est important de définir quant les tests unitaires sont écrits. Plusieurs mises en oeuvre sont possibles :

- Ecrire les tests juste après avoir écrit une méthode
- Idéalement, écrire les tests avant le code à tester
- Ecrire les tests, écrire le code pour faire échouer les tests, vérifier que les tests échouent, corriger le code, vérifier que les tests sont OK

Il faut développer de préférences les tests unitaires le plus tôt possible. Dans une approche traditionnelle, juste après l'écriture de la méthode. Dans une approche TDD (Test Drive Development), avant l'écriture de la méthode.
Ceci possède plusieurs avantages par rapport à une écriture ultérieure de tests :

- permet de détecter des bugs le plus rapidement possible,
- coder des cas oubliés dans la méthode,
- s'assurer que les tests unitaires sont écrits,
- écrire du code testable évitant ainsi un refactoring parfois conséquant
- ...

Il est préférable d'appliquer trois règles avec les tests unitaires :

- tester le plus possible : afin d'augmenter les chances de découvrir des bugs
- tester le plus tôt possible : plus les tests sont faits tôt plus les bugs sont rapidement détectés
- tester le souvent possible : en les automatisant et si possible en les intégrant dans un processus d'intégration continue

Un test unitaire devrait respecter certains principes :

- le test doit être le plus petit et le plus simple possible
- chaque test doit être isolé : un test ne doit pas dépendre d'un autre. Ceci permet aussi de garantir qu'une modification d'un test n'aura pas d'impact sur un autre
- pour pouvoir être facilement exécutés régulièrement, les tests unitaires doivent être automatisés

Le rôle des tests unitaires est d'automatiser des tests sur des unités de code les plus petites possibles, généralement une méthode. Cependant le code d'une méthode peut avoir besoin d'autres objets ou de ressources externes.

Plus le code à tester va avoir de dépendances plus il sera difficile à tester. Il faut donc minimiser ces dépendances en utilisant plusieurs solutions :

- utilisation de design patterns
- utilisation d'objets de type mock
- éviter de faire appels à la base de données dans les cas de tests
- ...

Un test unitaire doit impérativement se faire de façon isolée donc sans dépendre d'autres tests ni de requérir les dépendances utilisées par la fonctionnalité en cours de test. Le but d'un test unitaire est de tester les traitements de la fonctionnalité et non de tester les interactions qu'elle peut avoir avec ces dépendances.

Un test unitaire doit obligatoirement être répétable pour obtenir toutes ces lettres de noblesse. Cela permet de capitaliser les tests unitaires non seulement pour les tests unitaires mais aussi pour les tests de non regression.

Chaque cas de tests doit être autonome et ne doit donc pas dépendre d'un ou plusieurs autres cas de tests.

Il est pratique de définir et d'utiliser des conventions de nommages pour les classes de tests. Certaines sont imposées par le framework de tests utilisés : dans ce cas leurs mises en oeuvre est obligatoire. Dans les autres cas, il est préférable de définir ces propres conventions et de les mettre en oeuvre, par exemple :

- préfixer les classes de tests par Test suivi du nom de la classe testée et les mettre dans package dédié dont le nom correspond au nom du package des classes testés préfixé par test
- mettre les classes de test dans le même package que les classes à tester en les suffixant par Test. Cela permet entre autre de facilement identifier les classes sans classes de tests. Ant peut être utilisé pour filtrer les classes à inclure dans la génération des livrables
- écrire une classe de tests par classe testée

78.1.3. Les difficultés lors de la mise en oeuvre de tests unitaires

Plusieurs difficultés sont rencontrées lors de la mise en oeuvre de tests unitaires :

- réticences à la mise en oeuvre
- difficultés de rédaction et de codage
- couverture du code testé
- temps nécessaire à la rédaction des cas tests
- véracité des cas de tests
- temps nécessaire à la maintenance des cas de tests
- les cas de tests doivent être répétables
- il n'y a pas que le code qui doit être testé, il est aussi nécessaire de tester les valeurs de certaines ressources (base de données, fichiers, ...)
- ...

Lorsque l'on parle aux développeurs de rédiger des tests unitaires, il est fréquent d'obtenir des réticences avec des justifications futiles :

- "Je n'ai pas le temps",
- "Je ne sais pas les écrire",
- "Ce n'est pas mon job",
- "Je ne fais jamais de bugs",
- ...

Dans la plupart des cas, il est plus difficile d'écrire les tests que d'écriture le code à tester. Ainsi, l'écriture du code d'une application est un art mais l'écriture de tests pour ce code est un art encore plus complexe. De ce fait, la rédaction des cas de tests est fréquemment confiée à des développeurs expérimentés ou dédiés à cette activité.

Les tests unitaires doivent évoluer avec le code de l'application. Il est donc très important que le code des tests unitaires soit simple, compréhensible et maintenable.

La mise en oeuvre de tests unitaires automatisés augmente la fiabilité du code mais elle ne peut pas offrir une garantie à 100% pour plusieurs raisons :

- la couverture du code testé ne peut généralement pas être totale
- il est impossible de couvrir tous les cas de tests
- les tests unitaires peuvent contenir eux même des bugs

Il n'est pas possible de couvrir tous les cas possibles avec des cas de tests unitaires. Il est donc nécessaire de déterminer quelles classes possèderont des tests unitaires, de maximiser le nombre de ces classes testées, de définir les cas de tests de chaque classes et de maximiser le nombre de ces cas.

Une des grandes difficultés lors de la rédaction de cas de tests est de s'assurer qu'un maximum de cas de tests est implémenté. Il ne faut surtout pas se contenter de ne tester que les cas de fonctionnement standards mais aussi couvrir un maximum de cas de fonctionnement anormaux (données invalides, levée d'exceptions, tests aux limites, ...).

Les tests unitaires sont composés de code qui peut contenir comme toute portion de code des bugs.

Généralement, les tests unitaires possèdent des dépendances vers des ressources externes (fichiers, bases de données, bibliothèques tiers, connexions réseau, ...). L'utilisation de ce ressources dans les tests unitaires doit être éviter car généralement elles limitent la répétitabilités des tests et entraîne un surcoût dans le temps d'éxécution des tests unitaires.

Malgré ces difficultés, les tests unitaires automatisés ne doivent pas être occultés car ils peuvent améliorer de façon significative la qualité et la fiabilité du code lors de son écrire et surtout de sa maintenance.

78.1.4. Des best practices

La rédaction des tests unitaires devrait suivre quelques recommandations :

- le nom des tests devrait permettre de facilement fournir une indication sur le but du test
- il est préférable de n'avoir qu'un seul assert par test car un test ne devrait avoir qu'une seule raison d'échouer
- le code des tests unitaires doit être maintenu au même titre que le code qu'il teste : la même attention doit être portée dans leur écrire (respect des normes, commentaires, refactoring, ...)
- stocker les tests unitaires dans un package dédié dont le nom est celui du package de la classe à tester avec le prefixe test

Chaque test unitaire doit s'exécuter le plus rapidement possible : le nombre de tests unitaires va croître au fur et à mesure des développements donc le temps d'exécution des tests va croître lui aussi.

Il est préférable d'inclure l'exécution des tests unitaires dans un processus d'intégration continue.

Il faut conserver les cas de tests le plus simple possible. Par exemple, pour le test d'une méthode qui additionne deux nombres, il est préférable pour tester le cas standard qui utilise de petits nombres plutôt que d'utiliser de grands nombres. La véracité du test est la même mais le test est plus facile à comprendre et à vérifier.

Chaque test doit correspondre à un cas de test unique. Il est préférable de n'avoir qu'un seul test dans un cas de test, soit une seule instruction de type assert. Ceci rendra le code du test plus simple et facilitera le calcul de métriques lors de l'exécution de tests.

Le test d'un constructeur nécessitent généralement l'invocation de getter et setter pour vérifier les valeurs des paramètres fournis au constructeur qui sont généralement utilisées pour initialiser directement ou indirectement des champs de l'objet.

Il n'est pas toujours facile de rendre les tests d'une méthode indépendants de l'utilisation d'autres méthodes. Par exemple, il est difficile de tester un setter sans faire appel au getter de la propriété correspondante.

Pour tester des méthodes privées, il faut tester les méthodes qui font appels à ces méthodes privées.

Il est aussi généralement non trivial, de tester une méthode qui n'a pas de paramètre de retour. Ces méthodes effectuent généralement des modifications sur des éléments internes ou externes à la classe. Il faut alors capturer le résultat de ces modifications pour pouvoir réaliser les tests.

Il ne faut pas remonter dans le gestionnaire de source du code dont un ou plusieurs tests unitaires échouent.

Il ne faut pas hésiter à enrichir les tests avec de nouveaux cas ou créer des cas de tests pour des classes qui n'en ont pas. Le code d'une application et le code des tests unitaires doivent évoluer dans une optique d'améliorations continues.

A chaque maintenance dans le code, les tests unitaires doivent être exécutés et maintenus eux aussi au besoin. Il ne faut surtout pas livrer du code dont au moins un test unitaire échoue quelques soient les raisons.

78.2. Les frameworks et outils de tests

De nombreux frameworks et outils open source sont proposés pour faciliter la mise en oeuvre des tests

- frameworks de tests unitaires et leur extensions
- frameworks pour le mocking
- outils de tests de charge
- outils d'analyse de couverture du test
- ...

78.2.1. Les frameworks pour les tests unitaires

Plusieurs frameworks open source sont utilisables dans le monde Java notamment :

- JUnit : C'est le plus ancien et le plus répandu ce qui en fait un standard de facto
- TestNG :

JUnit est à l'origine de plusieurs frameworks similaires pour différentes plateformes ou langages notamment nUnit (.Net), dUnit (Delphi), cppUnit (C++), ... Tous ces frameworks sont regroupés dans une famille nommée xUnit.

78.2.2. Les frameworks pour le mocking

Généralement les tests unitaires de code d'une application, notamment celles développées en couches, nécessitent l'utilisation d'objets de type mock pour permettre de se concentrer sur le test du code de la méthode en minimisant les effets de bord liés aux autres objets utilisés dans le code.

L'utilisation de ces frameworks est détaillé dans le chapitre «Les objets de type mock» .

78.2.3. Les extensions de JUnit

JUnit est utilisé dans un certain nombre de projets qui proposent d'étendre ses fonctionnalités :

* Cactus : un framework open source de tests pour des composants serveur J2EE
* JunitReport : une tache Ant pour générer un rapport des tests effectués avec JUnit sous Ant
* JWebUnit : un framework open source de tests pour des applications web
* StrutsTestCase : extension de JUnit pour les tests d'application utilisant Struts 1.0.2 et 1.1
* XMLUnit : extension de JUnit pour les tests sur des documents XML

Cactus est un framework open source de tests pour des composants serveur. Il est développé en s'appuyant sur JUnit par le projet Jakarta du groupe Apache.

La dernière version de Cactus peut être téléchargée sur le site http://jakarta.apache.org/cactus/.

78.2.4. Les outils de tests de charge

Apache JMeter est l'outil de tests de charge pour tout ce qui repose sur le protocole http le plus répandu.

SoapUI est particulièrement adapté pour les tests et les tests de charges de services web.

78.2.5. Les outils d'analyse de couverture de tests

Des outils sont proposés pour vérifier le taux de couverture des cas de tests vis-à-vis du code (test coverage analyser).

Le but de ces outils est de faciliter la détermination des fonctionnalités qui possèdent des tests et par conséquent permettent de déterminer quelles portions du code ne sont pas testées du tout ou insuffisamment testées.

Plusieurs outils open source existent notamment :

* Cobertura
* JCoverage
* Jester
* CodeCover

79. JUnit

JUnit est un framework open source pour le développement et l'exécution de tests unitaires automatisables. Le principal intérêt est de s'assurer que le code répond toujours aux besoins même après d'éventuelles modifications. Plus généralement, ce type de tests est appelé tests unitaires de non régression.

JUnit a été initialement développé par Erich Gamma et Kent Beck.

JUnit propose :

- Un framework pour le développement des tests unitaires reposant sur des assertions qui testent les résultats attendus
- Des applications pour permettre l'exécution des tests et afficher les résultats

Le but est d'automatiser les tests. Ceux ci sont exprimés dans des classes sous la forme de cas de tests avec leurs résultats attendus. JUnit exécute ces tests et les comparent avec ces résultats.

Cela permet de séparer le code de la classe, du code qui permet de la tester. Souvent pour tester une classe, il est facile de créer une méthode main() qui va contenir les traitements de tests. L'inconvénient est que ce code "superflu" aux traitements proprement dit est qu'il soit inclus dans la classe. De plus, son exécution doit se faire manuellement.

La rédaction de cas de tests peut avoir un effet immédiat pour détecter des bugs mais surtout elle a un effet à long terme qui facilite la détection d'effets de bords lors de modifications.

Les cas de tests sont regroupés dans des classes Java qui contiennent une ou plusieurs méthodes de tests. Ces cas de tests peuvent être regroupés sous la forme de suites de tests. Les cas de tests peuvent être exécutés individuellement ou sous la forme de suites de tests.

JUnit permet le développement incrémental d'une suite de tests.

Avec JUnit, l'unité de test est une classe dédiée qui regroupe des cas de tests. Ces cas de tests exécutent les tâches suivantes :

- création d'une instance de la classe et de tout autre objet nécessaire aux tests
- appel de la méthode à tester avec les paramètres du cas de test
- comparaison du résultat attendu avec le résultat obtenu : en cas d'échec, une exception est levée

JUnit est particulièrement adapté pour être utilisé avec la méthode eXtreme Programming puisque cette méthode préconise entre autre l'automatisation des tâches de tests unitaires qui ont été définis avant l'écriture du code.

La version utilisée dans ce chapitre est la 3.8.1 sauf dans la section dédiée à la version 4 de JUnit.

La page officielle est à l'url : http://junit.org.

La dernière version de JUnit peut être téléchargée sur le site www.junit.org. Pour l'installer, il suffit de décompresser l'archive dans un répertoire du système.

Pour pouvoir utiliser JUnit, il faut ajouter le fichier junit.jar au classpath.

Ce chapitre contient plusieurs sections :

- Un exemple très simple
- L'écriture des cas de tests
- L'exécution des tests
- Les suites de tests
- L'automatisation des tests avec Ant
- JUnit 4

79.1. Un exemple très simple

L'exemple utilisé dans cette section est la classe suivante :

Exemple :

```
public class MaClasse{

  public static int calculer(int a, int b) {
    int res = a + b;

    if (a == 0){
      res = b * 2;
    }

    if (b == 0) {
      res = a * a;
    }
    return res;
  }

}
```

Il faut compiler cette classe : javac MaClasse.java

Il faut ensuite écrire une classe qui va contenir les différents tests à réaliser par JUnit. L'exemple est volontairement simpliste en ne définissant qu'un seul cas de test.

Exemple :

```
import junit.framework.*;

public class MaClasseTest extends TestCase{

  public void testCalculer() throws Exception {

    assertEquals(2,MaClasse.calculer(1,1));
  }

}
```

Il faut compiler cette classe avec le fichier junit.jar qui doit être dans le classpath.

Enfin, il suffit d'appeler JUnit pour qu'il exécute la séquence de tests.

Exemple :

```
java -cp junit.jar;. junit.textui.TestRunner MaClasseTest
C:\java\testjunit>java -cp junit.jar;. junit.textui.TestRunner
  MaClasseTest
  .
  Time: 0,01
OK (1 test)
```

Attention : le respect de la casse dans le nommage des méthodes de tests est très important. Les méthodes de tests doivent obligatoirement commencer par test en minuscule car JUnit utilise l'introspection pour déterminer les méthodes à exécuter.

Exemple :

```
import junit.framework.*;

public class MaClasseTest extends TestCase{

  public void TestCalculer() throws Exception {

    assertEquals(2,MaClasse.calculer(1,1));
  }

}
```

L'utilisation de cette classe avec JUnit produit le résultat suivant :

Résultat :

```
C:\java\testjunit>java -cp junit.jar;. junit.textui.TestRunner
MaClasseTest
.F
Time: 0,01
There was 1 failure:
1) warning(junit.framework.TestSuite$1)junit.framework.AssertionFailedError: No
tests found in MaClasseTest

FAILURES!!!
Tests run: 1,  Failures: 1,  Errors: 0
```

79.2. L'écriture des cas de tests

JUnit propose un framework pour écrire les classes de tests.

Un test est une classe qui hérite de la classe TestCase. Par convention le nom de la classe de test est composé du nom de la classe suivi de Test.

Chaque cas de tests fait l'objet d'une méthode dans la classe de tests. Le nom de ces méthodes doit obligatoirement commencer par le suffixe test.

Chacune de ces méthodes contient généralement des traitements en trois étapes :

- Instanciation des objets requis
- Invocation des traitements sur les objets
- Vérification des résultats des traitements

Il est important de se souvenir lors de l'écriture de cas de tests que ceux-ci doivent être indépendants les uns des autres. JUnit ne garantit pas l'ordre d'exécution des cas de tests puisque ceux-ci sont obtenus par introspection.

Toutes les classes de tests avec JUnit héritent de la classe Assert.

79.2.1. La définition de la classe de tests

Pour écrire les cas de tests, il faut écrire une classe qui étende la classe junit.framework.TestCase. Le nom de cette classe est le nom de la classe à tester suivi par "Test".

Remarque : dans la version 3.7 de JUnit, une classe de tests doit obligatoirement posséder un constructeur qui attend un objet de type String en paramètre.

Exemple :

```
import junit.framework.*;

public class MaClasseTest extends TestCase{

  public MaClasseTest(String testMethodName) {
    super(testMethodName);
  }

  public void testCalculer() throws Exception {
    fail("Cas de test a ecrire");
  }
}
```

Dans cette classe, il faut écrire une méthode dont le nom commence par "test" en minuscule suivi du nom du cas de test (généralement le nom de la méthode à tester). Chacune de ces méthodes doit avoir les caractéristiques suivantes :

- elle doit être déclarée public
- elle ne doit renvoyer aucune valeur
- elle ne doit pas posséder de paramètres.

Par introspection, JUnit va automatiquement rechercher toutes les méthodes qui respectent cette convention.

Le respect de ces règles est donc important pour une bonne exécution des tests par JUnit.

Exemple : la méthode commence par T et non t

```
import junit.framework.*;

public class MaClasseTest extends TestCase{

  public void TestCalculer() throws Exception {

    // assertEquals(2,MaClasse.calculer(1,1));
    fail("Cas de test a ecrire");
  }

}
```

Résultat :

```
C:\>java -cp junit.jar;. junit.textui.TestRunner
MaClasseTest
.F
Time: 0,01
There was 1 failure:
1) warning(junit.framework.TestSuite$1)junit.framework.AssertionFailedError: No
tests found in MaClasseTest

FAILURES!!!
Tests run: 1,  Failures: 1,  Errors: 0
```

79.2.2. La définition des cas de tests

Chaque classe de tests doit avoir obligatoirement au moins une méthode de test sinon une erreur est remontée par JUnit.

La découverte des méthodes de tests par Junit repose sur l'introspection : JUnit recherche les méthodes qui débutent par test, n'ont aucun paramètre et ne retourne aucune valeur. Ces méthodes peuvent lever des exceptions qui sont automatiquement capturées par JUnit qui remonte alors une erreur et donc un échec du cas de tests.

Dès qu'un test échoue, l'exécution de la méthode correspondante est interrompue et JUnit passe à méthode suivante.

La classe suivante sera utilisée dans les exemples de cette section :

Exemple :

```
public class MaClasse2{
  private int a;
  private int b;

  public MaClasse2(int a, int b) {
    this.a = a;
    this.b = b;
  }

  public int getA() {
    return a;
  }

  public int getB() {
    return b;
  }

  public void setA(int unA) {
    this.a = unA;
  }

  public void setB(int unB) {
    this.b = unB;
  }

  public int calculer() {
    int res = a + b;

    if (a == 0){
      res = b * 2;
    }

    if (b == 0) {
      res = a * a;
    }
    return res;
  }

  public int sommer() throws IllegalStateException {
    if ((a == 0) && (b==0)) {
      throw new IllegalStateException("Les deux valeurs sont nulles");
    }
    return a+b;
  }
}
```

Avec JUnit, la plus petite unité de tests est l'assertion dont le résultat de l'expression booléenne indique un succès ou une erreur.

Les cas de tests utilisent des affirmations (assertion en anglais) sous la forme de méthodes nommées assertXXX() proposées par le framework. Il existe de nombreuses méthodes de ce type qui sont héritées de la classe junit.framework.Assert :

Méthode	Rôle
assertEquals()	Vérifier l'égalité de deux valeurs de type primitif ou objet (en utilisant la méthode equals()). Il existe de nombreuses surcharges de cette méthode pour chaque type primitif, pour un objet de type Object et pour un objet de type String
assertFalse()	Vérifier que la valeur fournie en paramètre est fausse
assertNull()	Vérifier que l'objet fourni en paramètre soit null
assertNotNull()	Vérifier que l'objet fourni en paramètre ne soit pas null
assertSame()	Vérifier que les deux objets fournis en paramètre font référence à la même entité Exemples identiques : assertSame("Les deux objets sont identiques", obj1, obj2); assertTrue("Les deux objets sont identiques ", obj1 == obj2);
assertNotSame()	Vérifier que les deux objets fournis en paramètre ne font pas référence à la même entité
assertTrue()	Vérifier que la valeur fournie en paramètre est vraie

Bien qu'il serait possible de n'utiliser que la méthode assertTrue(), les autres méthodes assertXXX() facilite l'expression des conditions de tests.

Chacune de ces méthodes possède une version surchargée qui accepte un paramètre supplémentaire sous la forme d'une chaîne de caractères indiquant un message qui sera affiché en cas d'échec du cas de test. Le message devrait décrire le cas de tests évalué à true.

L'utilisation de cette version surchargée est recommandée car elle facilite l'exploitation des résultats des cas de tests.

Exemple :

```
import junit.framework.*;

public class MaClasse2Test extends TestCase{

  public void testCalculer() throws Exception {
    MaClasse2 mc = new MaClasse2(1,1);
    assertEquals(2,mc.calculer());
  }

}
```

L'ordre des paramètres contenant la valeur attendue et la valeur obtenue est important pour obtenir un message d'erreur fiable en cas d'échec du cas de test. Quelque soit la surcharge utilisée l'ordre des deux valeurs à tester est toujours la même : c'est toujours la valeur attendue qui précède la valeur courante.

La méthode fail() permet de forcer le cas de test à échouer. Une version surchargée permet de préciser un message qui sera affiché.

Il est aussi souvent utile lors de la définition des cas de tests de devoir tester si une exception est levée lors de l'exécution des traitements.

Exemple :

```
import junit.framework.*;

public class MaClasse2Test extends TestCase{

  public void testSommer() throws Exception {
    MaClasse2 mc = new MaClasse2(0,0);
    mc.sommer();
```

```
  }
}
```

Résultat :

```
C:\>java -cp junit.jar;. junit.textui.TestRunner MaClasse2Test
.E
Time: 0,01
There was 1 error:
1) testSommer(MaClasse2Test)java.lang.IllegalStateException: Les deux valeurs so
nt nulles
        at MaClasse2.sommer(MaClasse2.java:42)
        at MaClasse2Test.testSommer(MaClasse2Test.java:31)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.
java:39)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAcces
sorImpl.java:25)

FAILURES!!!
Tests run: 2,  Failures: 0,  Errors: 1
```

Avec JUnit, pour réaliser de tels cas de tests, il suffit d'appeler la méthode avec les conditions qui doivent lever une exception, d'encapsuler cet appel dans un bloc try/catch et d'appeler la méthode fail() si l'exception désirée n'est pas levée.

Exemple :

```
import junit.framework.*;

public class MaClasse2Test extends TestCase{

  public void testSommer() throws Exception {
    MaClasse2 mc = new MaClasse2(1,1);

    // cas de test 1
    assertEquals(2,mc.sommer());

    // cas de test 2
    try {
      mc.setA(0);
      mc.setB(0);
      mc.sommer();
      fail("Une exception de type IllegalStateException aurait du etre levee");
    } catch (IllegalStateException ise) {
    }

  }
}
```

79.2.3. L'initialisation des cas de tests

Il est fréquent que les cas de tests utilisent une instance d'un même objet ou nécessitent l'usage de ressources particulières telles qu'une instance d'une classe pour l'accès à une base de données par exemple.

Pour réaliser ces opérations de création et de destruction d'objets, la classe TestCase propose les méthodes setUp() et tearDown() qui sont respectivement appelées avant et après l'appel de chaque méthode contenant un cas de test.

Il suffit simplement de redéfinir en fonction de ces besoins ces deux méthodes.

Cette section va tester le bean ci-dessous

Exemple :

```
package com.jmdoudoux.test.junit;

public class Personne {

  private String nom;

  private String prenom;

  public Personne() {
    super();
  }

  public Personne(String nom, String prenom) {
    super();
    this.nom = nom;
    this.prenom = prenom;
  }

  public String getNom() {
    return nom;
  }

  public void setNom(String nom) {
    this.nom = nom;
  }

  public String getPrenom() {
    return prenom;
  }

  public void setPrenom(String prenom) {
    this.prenom = prenom;
  }

}
```

Le plus simple est de définir un membre privé du type dont on a besoin et de créer une instance de ce type dans la méthode setUp().

Il est important de se souvenir que la méthode setUp() est invoquée systématiquement avant l'appel de chaque méthode de tests. Sa mise en oeuvre n'est donc requise que si toutes les méthodes de tests ont besoin de créer une instance d'un même type ou d'exécuter un même traitement.

Exemple :

```
package com.jmdoudoux.test.junit;

import junit.framework.TestCase;

public class PersonneTest extends TestCase {

  private Personne personne;

  public PersonneTest(String name) {
    super(name);
  }

  protected void setUp() throws Exception {
    super.setUp();
    personne = new Personne("nom1","prenom1");
  }

  protected void tearDown() throws Exception {
    super.tearDown();
    personne = null;
  }

  public void testPersonne() {
    assertNotNull("l'instance est créée", personne);
  }
```

```
  public void testGetNom() {
    assertEquals("est ce que nom est correct", "nom1", personne.getNom());
  }

  public void testSetNom() {
    personne.setNom("nom2");
    assertEquals("est ce que nom est correct", "nom2", personne.getNom());
  }

  public void testGetPrenom() {
    assertEquals("est ce que prenom est correct", "prenom1", personne.getPrenom());
  }

  public void testSetPrenom() {
    personne.setPrenom("prenom2");
    assertEquals("est ce que prenom est correct", "prenom2", personne.getPrenom());
  }

}
```

Ceci évite de créer l'instance dans chaque méthode de tests et simplifie donc l'écriture des cas de tests.

Dans l'exemple la méthode tearDown() remet à null l'instance créé : ceci n'est pas une obligation d'autant que le temps des traitements réalisés durant les tests est normalement négligeable. La méthode tearDown() peut cependant avoir un grand intérêt pour par exemple libérer des ressources comme une connexion à une base de données initiée dans la méthode setUp().

Pour des besoins particuliers, il peut être nécessaire d'exécuter du code une seule fois avant l'exécution des cas de tests et/ou exécuter du code une fois tous les cas des tests exécutés.

JUnit propose pour cela la classe junit.Extensions.TestSetup qui propose la mise en oeuvre du design pattern décorateur.

Exemple :

```
package com.jmdoudoux.test.junit;

import junit.extensions.TestSetup;
import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;

public class PersonneTest extends TestCase {

  private Personne personne;

  public PersonneTest(String name) {
    super(name);
  }

  protected void setUp() throws Exception {
    super.setUp();
    personne = new Personne("nom1", "prenom1");
  }

  protected void tearDown() throws Exception {
    super.tearDown();
    personne = null;
  }

...

  public static Test suite() {
    TestSetup setup = new TestSetup(new TestSuite(PersonneTest.class)) {
      protected void setUp() throws Exception {
        // code execute une seule fois avant l'exécution des cas de tests
        System.out
            .println("Appel de la methode setUp() de la classe de tests");
      }
```

```
      protected void tearDown() throws Exception {
        // code execute une seule fois après l'exécution de tous les cas de tests
        System.out
            .println("Appel de la methode tearDown() de la classe de tests");
      }
    };
    return setup;
  }

  public static void main(String[] args) {
      junit.textui.TestRunner.run(suite());
  }
}
```

Dans l'exemple ci-dessus, les méthodes setUp() et tearDown() de la classe PersonneTest seront toujours invoquées respectivement avant et après chaque exécution d'un cas de tests.

79.2.4. Le test de la levée d'exceptions

Il est fréquent qu'une méthode puisse lever une ou plusieurs exceptions durant son exécution. Il faut prévoir des cas de tests pour vérifier que dans les conditions adéquates une exception attendue est bien levée.

Exemple :

```
package com.jmdoudoux.test.junit;

public class Personne {

  private String nom;

  private String prenom;

  public Personne() {
    super();
  }

  public Personne(String nom, String prenom) {
    super();
    this.nom = nom;
    this.prenom = prenom;
  }

  public String getNom() {
    return nom;
  }

  public void setNom(String nom) {
    if (nom == null) {
      throw new IllegalArgumentException("la propriété nom ne peut pas être null");
    }
    this.nom = nom;
  }

  public String getPrenom() {
    return prenom;
  }

  public void setPrenom(String prenom) {
    this.prenom = prenom;
  }

}
```

Pour effectuer la vérification de la levée d'une exception, il faut inclure l'invocation de la méthode dans un bloc try/catch et faire appel à la méthode fail() si l'exception n'est pas levée.

Exemple :

```
package com.jmdoudoux.test.junit;

import junit.framework.TestCase;

public class PersonneTest extends TestCase {

  private Personne personne;

  public PersonneTest(String name) {
    super(name);
  }

  protected void setUp() throws Exception {
    super.setUp();
    personne = new Personne("nom1","prenom1");
  }

  protected void tearDown() throws Exception {
    super.tearDown();
    personne = null;
  }

  public void testPersonne() {
    assertNotNull("l'instance est créée", personne);
  }

  public void testGetNom() {
    assertEquals("est ce que nom est correct", "nom1", personne.getNom());
  }

  public void testSetNom() {
    personne.setNom("nom2");
    assertEquals("est ce que nom est correct", "nom2", personne.getNom());
    try {
      personne.setNom(null);
      fail("IllegalArgumentException non levée avec la propriété nom à null");
    } catch (IllegalArgumentException iae) {
      // ignorer l'exception puisque le test est OK (l'exception est levée)
    }
  }

  public void testGetPrenom() {
    assertEquals("est ce que prenom est correct", "prenom1", personne.getPrenom());
  }

  public void testSetPrenom() {
    personne.setPrenom("prenom2");
    assertEquals("est ce que prenom est correct", "prenom2", personne.getPrenom());
  }

}
```

Attention : une erreur courante lorsque l'on code ces premiers tests unitaires est d'inclure les invocations de méthodes dans des blocs try/catch. Leur utilisation doit être uniquement réservée comme dans l'exemple précédant. Dans tous les autres cas, il faut laisser l'exception se propager : dans ce cas, JUnit va automatiquement reporter un échec du test. Il est en particulier inutile d'utiliser un bloc try/catch et de faire appel à la méthode fail() dans le catch puisque JUnit le fait déjà.

79.2.5. L'héritage d'une classe de base

Il est possible de définir une classe de base qui servira de classe mère à d'autres classes de tests notamment en leur fournissant des fonctionnalités communes.

JUnit n'impose pas qu'une classe de tests dérive directement de la classe TestCase. Ceci est particulièrement pratique lorsque l'on souhaite que certaines initialisations ou certains traitements soit systématiquement exécutés (exemple

chargement d'un fichier de configuration, ...).

Il est par exemple possible de faire des initialisations dans le constructeur de la classe mère et invoquer ce constructeur dans les constructeurs des classe filles.

79.3. L'exécution des tests

JUnit propose trois applications différentes nommées TestRunner pour exécuter les tests en mode ligne de commande ou application graphique :

- une application console : junit.textui.TestRunner qui est très rapide et adaptée à une intégration dans un processus de générations automatiques.
- une application graphique avec une interface Swing : junit.swingui.TestRunner
- une application graphique avec une interface AWT : junit.awtui.TestRunner

Quelque soit l'application utilisée, les entités suivantes doivent être incluses dans le classpath :

- le fichier junit.jar
- classes à tester et les classes des cas de tests
- les classes et bibliothèques dont toutes ces classes dépendent

Suite à l'exécution d'un cas de test, celui ci peut avoir un des trois états suivants :

- échoué : une exception de type AssertionFailedError est levée
- en erreur : une exception non émise par le framework et non capturée a été levée dans les traitements
- passé avec succès

L'échec d'un seul cas de test entraîne l'échec du test complet.

L'échec d'un cas de test peut avoir plusieurs origines :

- le cas de test contient un ou plusieurs bugs
- le code à tester contient un ou plusieurs bugs
- le cas de test est mal défini
- une combinaison des cas précédents simultanément

79.3.1. L'exécution des tests dans la console

L'utilisation de l'application console nécessite quelques paramètres lors de son utilisation :

Résultat :

```
C:\>java -cp junit.jar;. junit.textui.TestRunner MaClasseTest
```

Le seul paramètre obligatoire est le nom de la classe de tests. Celle ci doit obligatoirement être sous la forme pleinement qualifiée si elle appartient à un package.

Résultat :

```
C:\>java -cp junit.jar;. junit.textui.TestRunner com.jmdoudoux.test.junit.MaClasseTest
```

Il est possible de faire appel au TestRunner dans une application en utilisant sa méthode run() en lui passant en paramètre un objet de type Class qui encapsule la classe de tests à exécuter.

Résultat :

```
public class TestJUnit1 {
    public static void main(String[] args) {
        junit.textui.TestRunner.run(MaClasseTest.class);
    }
}
```

Le TestRunner affiche le résultat de l'exécution des tests dans la console.

La première ligne contient un caractère point pour chaque test exécuté. Lorsque de nombreux tests sont exécutés cela permet de suivre la progression.

Le temps total d'exécution en seconde est ensuite affiché sur la ligne "Time:"

Enfin, un résumé des résultats de l'exécution est affiché.

Résultat :

```
.
Time: 0,078

OK (1 test)
```

En cas d'erreur, la première ligne contient un F à la suite du caractère point correspondant au cas de test en échec.

Le résumé de l'exécution affiche le détail de chaque cas de tests qui a échoué.

Résultat :

```
.F
Time: 0,063
There was 1 failure:
1) testCalculer(com.jmdoudoux.test.junit.MaClasseTest)junit.framework.AssertionFailedError:
 expected:<3> but was:<2>
        at com.jmdoudoux.test.junit.MaClasseTest.testCalculer(MaClasseTest.java:9)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
        at com.jmdoudoux.test.junit.MaClasseTest.main(MaClasseTest.java:13)

FAILURES!!!
Tests run: 1,  Failures: 1,  Errors: 0
```

Les cas en échec (failures) correspondent à une vérification faite par une méthode assertXXX() qui a échoué.

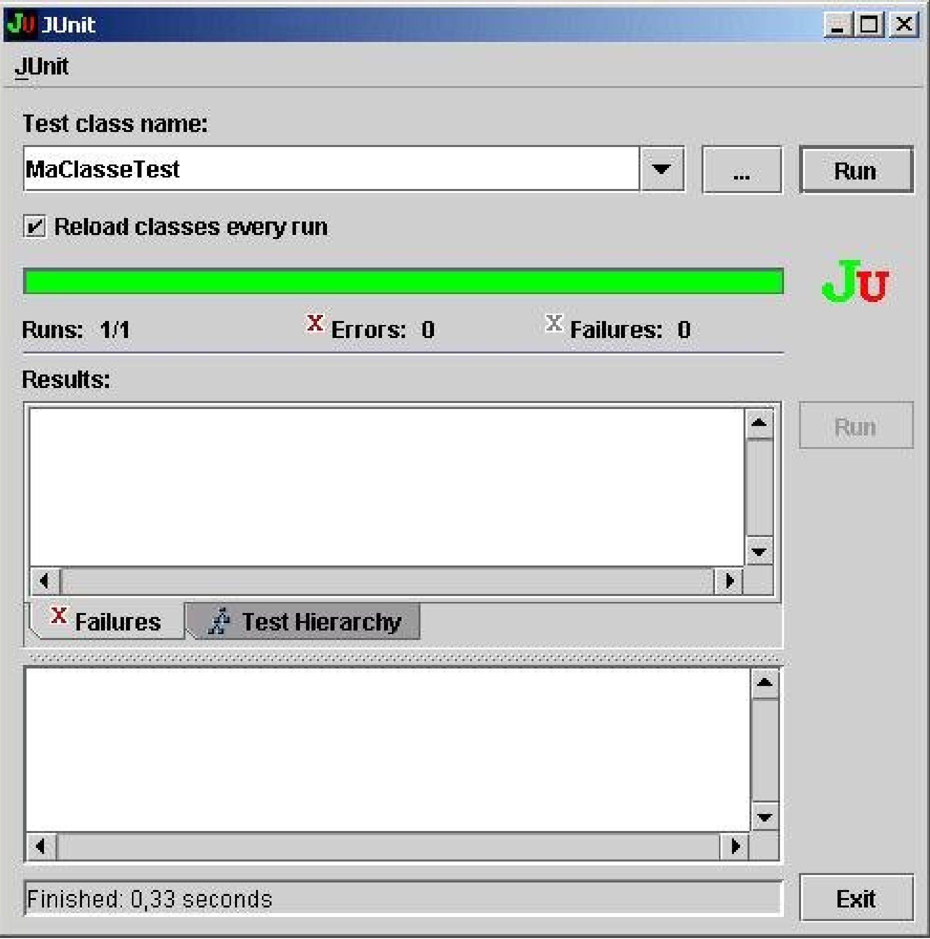
Les cas en erreur (errors) correspondent à la levée inattendue d'une exception lors de l'exécution du cas de test.

79.3.2. L'exécution des tests dans une application graphique

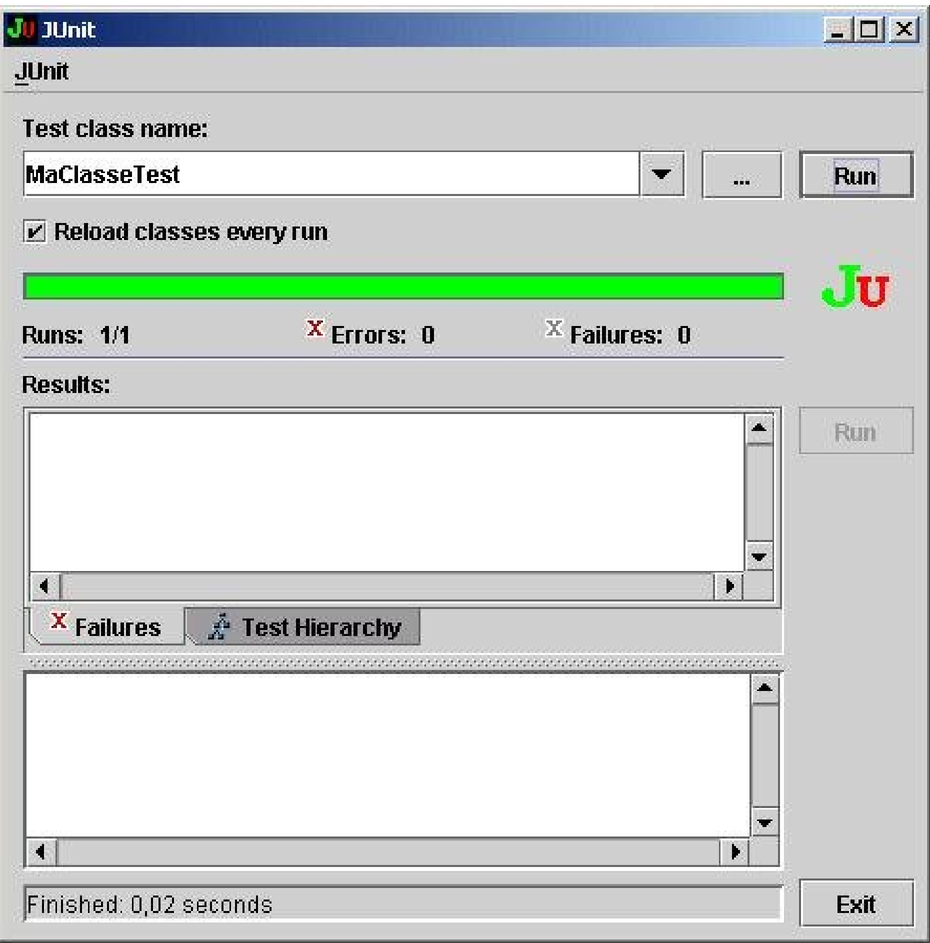
Pour utiliser des classes de tests avec ces applications graphiques, il faut obligatoirement que les classes de tests et toutes celles dont elles dépendent soient incluses dans le CLASSPATH. Elles doivent obligatoirement être sous la forme de fichier .class non incluses dans un fichier jar.

Exemple :

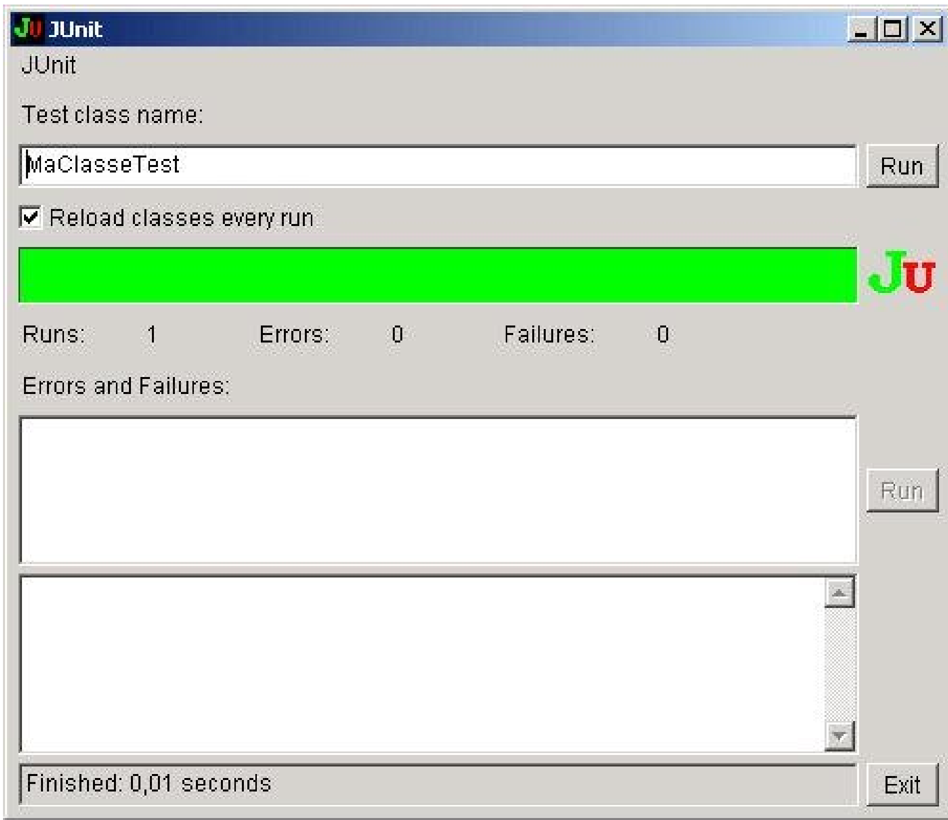
```
C:\java\testjunit>java -cp junit.jar;. junit.swingui.TestRunner MaClasseTest
```



Il suffit de cliquer sur le bouton "Run" pour lancer l'exécution des tests.



C:\java\testjunit>java -cp junit.jar;. junit.awtui.TestRunner MaClasseTest



La case à cocher "Reload classes every run" indique à JUnit de recharger les classes à chaque exécution. Ceci est très pratique car cela permet de modifier les classes et de laisser l'application de tests ouverte.

Si un ou plusieurs tests échouent la barre de résultats n'est plus verte mais rouge. Dans ce cas, le nombre d'erreurs et d'échecs est affiché ainsi que leur liste complète. Il suffit d'en sélectionner un pour obtenir le détail de la raison du problème.

Il est aussi possible de ne réexécuter que le cas sélectionné.

79.3.3. L'exécution d'une classe de tests

Il est possible de définir une classe main() dans une classe de tests qui va se charger d'exécuter les tests.

Exemple :

```
public class MaClasseTest extends TestCase {
  ...
  public static void main(String[] args) {
    junit.textui.TestRunner.run(new TestSuite(MaClasseTest.class));
  }
}
```

79.3.4. L'exécution répétée d'un cas de test

JUnit propose la classe junit.extensions.RepeatedTest qui permet d'exécuter plusieurs fois la même suite de tests.

Le constructeur de cette classe attend en paramètre une instance de la suite de tests et le nombre de répétitions de l'exécution de la suite de tests.

Exemple :

```
package com.jmdoudoux.test.junit;

import junit.extensions.RepeatedTest;
import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;

public class PersonneTest extends TestCase {

  private Personne personne;

  public PersonneTest(String name) {
    super(name);
  }

...

  public static Test suite() {
    return new RepeatedTest(new TestSuite(PersonneTest.class), 5);
  }

  public static void main(String[] args) {
      junit.textui.TestRunner.run(suite());
  }
}
```

79.3.5. L'exécution concurrente de tests

JUnit propose la classe junit.extensions.ActiveTestSuite qui permet d'exécuter plusieurs suites de tests chacune dans un thread dédié. Ainsi l'exécution des suites de tests se fait de façon concurrente.

Exemple :

```
package com.jmdoudoux.test.junit;

import junit.extensions.ActiveTestSuite;
import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;

public class PersonneTest extends TestCase {

  private Personne personne;

  public PersonneTest(String name) {
    super(name);
  }

...

  public static Test suite() {
    TestSuite suite = new ActiveTestSuite();
    suite.addTest(new TestSuite(PersonneTest.class));
    suite.addTest(new TestSuite(PersonneTest.class));
    suite.addTest(new TestSuite(PersonneTest.class));
    return suite;

  }

  public static void main(String[] args) {
    junit.textui.TestRunner.run(suite());
  }
}
```

L'ensemble de la suite de tests ne se termine que lorsque tous les threads sont terminés.

Même si cela n'est pas recommandé, la classe ActiveTestSuite peut être utilisée comme un outil de charge rudimentaire. Il est ainsi possible de combiner l'utilisation des classes ActiveTestsuite et RepeatedTest.

Exemple :

```
package com.jmdoudoux.test.junit;

import junit.extensions.ActiveTestSuite;
import junit.extensions.RepeatedTest;
import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;

public class PersonneTest extends TestCase {

  private Personne personne;

  public PersonneTest(String name) {
    super(name);
  }

...

  public static Test suite() {
    TestSuite suite = new ActiveTestSuite();
    suite.addTest(new RepeatedTest(new TestSuite(PersonneTest.class), 10));
    suite.addTest(new RepeatedTest(new TestSuite(PersonneTest.class), 20));
    return suite;

  }

  public static void main(String[] args) {
    junit.textui.TestRunner.run(suite());
  }
}
```

79.4. Les suites de tests

Les suites de tests permettent de regrouper plusieurs tests dans une même classe. Ceci permet l'automatisation de l'ensemble des tests inclus dans la suite et de préciser leur ordre d'exécution.

Pour créer une suite, il suffit de créer une classe de type TestSuite et d'appeler la méthode addTest() pour chaque classe de tests à ajouter. Celle ci attend en paramètre une instance de la classe de tests qui sera ajoutée à la suite. L'objet de type TestSuite ainsi créé doit être renvoyé par une méthode dont la signature doit obligatoirement être public static Test suite(). Celle ci sera appelée par introspection par le TestRunner.

Il peut être pratique de définir une méthode main() dans la classe qui encapsule la suite de tests pour pouvoir exécuter le TestRunner de la console en exécutant directement la méthode statique Run(). Ceci évite de lancer JUnit sur la ligne de commandes.

Exemple :

```
import junit.framework.*;

public class ExecuterLesTests {

  public static Test suite() {
    TestSuite suite = new TestSuite("Tous les tests");
    suite.addTestSuite(MaClasseTest.class);
    suite.addTestSuite(MaClasse2Test.class);

    return suite;
  }
```

```
  public static void main(String args[]) {
    junit.textui.TestRunner.run(suite());
  }
}
```

Deux versions surchargées des constructeurs permettent de donner un nom à la suite de tests.

Un constructeur de la classe TestSuite permet de créer automatiquement par introspection une suite de tests contenant tous les tests de la classe fournie en paramètre.

Exemple :

```
import junit.framework.*;

public class ExecuterLesTests2 {

  public static Test suiteDeTests() {
    TestSuite suite = new TestSuite(MaClasseTest.class,"Tous les tests");
    return suite;
  }

  public static void main(String args[]) {
    junit.textui.TestRunner.run(suiteDeTests());
  }
}
```

Pour éviter d'avoir à gérer une suite de tests, il est possible d'utiliser la tâche Ant optionnelle junit pour exécuter un ensemble de cas de tests en fonction d'un filtre sur le nom des classes.

Exemple :

```
...
<junit printsummary="yes" haltonfailure="yes">
  ...
  <batchtest fork="yes">
    <fileset dir="${src.dir}">
       <include name="**/Test*.java" />
    </fileset>
  </batchtest>
</junit>
...
```

Le détail de la mise en oeuvre de JUnit avec Ant est couvert dans la section suivante.

La méthode addTestSuite() permet d'ajouter à une suite une autre suite.

79.5. L'automatisation des tests avec Ant

L'automatisation des tests fait par JUnit au moment de la génération de l'application est particulièrement pratique. Ainsi Ant propose une tâche optionnelle dédiée nommée junit pour exécuter un TestRunner dans la console.

Pour pouvoir utiliser cette tâche, les fichiers junit.jar (fourni avec JUnit) et optional.jar (fourni avec Ant) doivent être accessibles dans le CLASSSPATH.

Cette tâche possède plusieurs attributs dont aucun n'est obligatoire et les principaux sont :

Attribut	Rôle	Valeur par défaut
printsummary	affiche un résumé statistique de l'exécution de chaque test	off

fork	exécution du TestRunner dans un JVM séparée	off
haltonerror	arrêt de la génération en cas d'erreur	off
haltonfailure	arrêt de la génération en cas d'échec d'un test	off
outfile	base du nom du fichier qui va contenir les résultats de l'exécution	

La tache <junit> peut avoir les éléments fils suivants : <jvmarg>, <sysproperty>, <env>, <formatter>, <test>, <batchtest>

L'élément <formatter> permet de préciser le format de sortie des résultats de l'exécution des tests. Il possède l'attribut type qui précise le format (les valeurs possibles sont : xml, plain ou brief) et l'attribut usefile qui précise si les résultats doivent être envoyés dans un fichier (les valeurs possibles sont : true ou false)

L'élément <test> permet de préciser un cas de test simple ou une suite de test selon le contenu de la classe précisée par l'attribut name. Cet élément possède de nombreux attributs et il est possible d'utiliser un élément fils de type <formatter> pour définir le format de sortie du test.

L'élément <batchtest> permet de réaliser toute une série de tests. Cet élément possède de nombreux attributs et il est possible d'utiliser un élément fils de type <formatter> pour définir le format de sortie des tests. Les différentes classes dont les tests sont à exécuter sont précisées par un élément fils <fileset>.

La tache <junit> doit être exécutée après la compilation des classes à tester.

Exemple : extrait d'un fichier build.xml pour Ant

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<project name="TestAnt1" default="all">
   <description>Génération de l'application</description>
     <property name="bin" location="bin"/>
        <property name="src" location="src"/>
        <property name="build" location="build"/>
        <property name="doc" location="${build}/doc"/>
        <property name="lib" location="${build}/lib"/>

        <property name="junit_path" value="junit.jar"/>

        ...
        <target name="test" depends="compil" description="Executer les tests avec JUnit">
           <junit fork="yes" haltonerror="true" haltonfailure="on" printsummary="on">
              <formatter type="plain" usefile="false" />
              <test name="ExecuterLesTests"/>
              <classpath>
                 <pathelement location="${bin}"/>
                 <pathelement location="${junit_path}"/>
              </classpath>
           </junit>
        </target>
        ...

</project>
```

Cet exemple exécute les tests de la suite de tests encapsulés dans la classe ExecuterLesTests

79.6. JUnit 4

JUnit version 4 est une évolution majeure depuis les quelques années d'utilisation de la version 3.8.

Un des grands bénéfices de cette version est l'utilisation des annotations introduites dans Java 5. La définition des cas de tests et des tests ne se fait donc plus sur des conventions de nommage et sur l'introspection mais sur l'utilisation d'annotations ce qui facilite la rédaction des cas de tests.

Une compatibilité descendante est assurée avec les suites de tests de JUnit 3.8.

JUnit 4 requiert une version 5 ou ultérieure de Java.

Le nom du package des classes de JUnit est différent entre la version 3 et 4 :

- les classes de Junit 3 sont dans le package junit.framework.
- les classes de Junit 4 sont dans le package org.junit.

79.6.1. La définition d'une classe de tests

Une classe de tests n'a plus l'obligation d'étendre la classe TestCase sous réserve d'utiliser les annotations définies par JUnit et d'utiliser des imports static sur les méthodes de la classe org.junit.Assert.

Exemple :

```
package com.jmdoudoux.test.junit4;

import org.junit.*;
import static org.junit.Assert.*;

public class MaClasse {

}
```

79.6.2. La définition des cas de tests

Les méthodes contenant les cas de tests n'ont plus d'obligation à utiliser la convention de nommage qui imposait de préfixer le nom des méthodes par test.

Avec JUnit 4, il suffit d'annoter la méthode avec l'annotation @Test.

Il est ainsi possible d'utiliser n'importe quelle méthode comme cas de test simplement en utilisant l'annotation @Test.

Exemple :

```
  @Test
  public void getNom() {
    assertEquals("est ce que nom est correct", "nom1", personne.getNom());
  }
```

Ceci permet d'utiliser le nom de méthode que l'on souhaite. Il est cependant conseillé de définir et d'utiliser une convention de nommage qui facilitera l'identification des classes de tests et des cas de tests. Il est par exemple possible de maintenir les conventions de nommage de JUnit 3.

L'annotation @Ignore permet de demander au framework d'ignorer un cas de tests. Les cas de tests dans ce cas sont marqués avec la lettre I lors de leur exécution en mode console.

Attention : l'utilisation de l'annotation @Ignore devrait être temporaire et justifié. Son utilisation ne doit pas devenir une solution à certains problèmes.

JUnit 4 inclut deux nouvelles surcharges de la méthode assertEquals() qui permettent de comparer deux tableaux d'objets. La comparaison se fait sur le nombre d'occurrences dans les tableaux et sur l'égalité de chaque objet d'un tableau dans l'autre tableau.

79.6.3. L'initialisation des cas des tests

JUnit 3 imposait une redéfinition des méthodes setUp() et TearDown() pour définir des traitements exécutés systématiquement avant et après chaque cas de test.

JUnit 4 propose simplement d'annoter la méthode exécutée avant avec l'annotation @Before et la méthode exécutée après avec l'annotation @After.

Exemple :

```
@Before
public void initialiser() throws Exception {
  personne = new Personne("nom1","prenom1");
}

@After
public void nettoyer() throws Exception {
  personne = null;
}
```

Il est possible d'annoter une ou plusieurs méthodes avec @Before ou @After. Dans ce cas, toutes les méthodes seront invoquées ou moment correspondant à leur annotation.

Il n'est pas nécessaire d'invoquer explicitement les méthodes annotées avec @Before et @After d'une classe mère. Tant que ces méthodes ne sont pas redéfinies, elles seront automatiquement invoquées lors de l'exécution des tests :

- les méthodes annotées avec @Before de la classe mère seront invoquées avant celles de la classe fille
- les méthodes annotées avec @After de la classe fille seront invoquées avant celles de la classe mère

JUnit 4 propose simplement d'annoter une ou plusieurs méthodes exécutées avant l'exécution du premier cas de tests avec l'annotation @BeforeClass et une ou plusieurs méthodes exécutées après l'exécution de tous les cas de test de la classe avec l'annotation @AfterClass.

Ces initialisations peuvent être très utiles notamment pour des connexions coûteuses à des ressources qu'il est préférable de ne réaliser qu'une seule fois plutôt qu'à chaque cas de tests. Ceci peut contribuer à améliorer les performances lors de l'exécution des tests.

79.6.4. Le test de la levée d'exceptions

Avec JUnit 3, pour vérifier la levée d'une exception dans un cas de test, il faut entourer l'appel du traitement dans un bloc try/catch et invoquer la méthode fail() à la fin du bloc try.

JUnit 4 propose une annotation pour faciliter la vérification de la lever d'une exception.

L'attribut expected de l'annotation @Test attend comme valeur la classe de l'exception qui devrait être levée.

Exemple :

```
@Test(expected=IllegalArgumentException.class)
public void setNom() {
  personne.setNom("nom2");
  assertEquals("est ce que nom est correct", "nom2", personne.getNom());
  personne.setNom(null);
}
```

Si lors de l'exécution du test l'exception du type précisée n'est pas levée (aucune exception levée ou une autre exception est levée) alors le test échoue.

Attention : l'utilisation de l'annotation ne permet que de vérifier que l'exception est levée. Pour vérifier des propriétés de l'exception, il est nécessaire d'utiliser le mécanisme utilisé avec JUnit 3 pour capturer l'exception et ainsi avoir accès aux

membres de son instance.

79.6.5. L'exécution des tests

Les applications graphiques AWT et Swing permettant l'exécution et l'affichage des résultats des cas de tests ne sont plus fournies avec JUnit 4.

JUnit laisse le soin de cette restitution aux IDE qui intègrent JUnit 4 comme par exemple Eclipse.

Une autre grande différence dans la façon d'exécuter les cas de tests avec JUnit 4 concerne le fait qu'il n'y a plus de différence entre un test échoué (échec d'une méthode assert() ou appel à la méthode fail()) et un test en erreur (une exception inattendue est levée).

Lors de l'exécution, si un avertissement de type "AssertionFailedError: No tests found in XXX" est fourni par JUnit c'est n'y a aucun cas de tests de fourni dans la classe (aucune méthode n'est annotées avec l'annotation @Test).

Dans une classe de tests, il est toujours possible de définir une classe main() qui permettent de demander l'exécution des cas de tests de la classe. Il faut invoquer la méthode main() de la classe org.junit.runner.JUnitCore.

Exemple :

```
package com.jmdoudoux.test.junit4;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotNull;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class PersonneTest {

  private Personne personne;

  @Before
  public void initialiser() throws Exception {
    personne = new Personne("nom1","prenom1");
  }

  @After
  public void nettoyer() throws Exception {
    personne = null;
  }

  @Test
  public void personne() {
    assertNotNull("l'instance est créée", personne);
  }

...

  public static void main(String[] args) {
    org.junit.runner.JUnitCore.main("com.jmdoudoux.test.junit4.PersonneTest");
  }
}
```

79.6.6. Un exemple de migration de JUnit 3 vers JUnit 4

La section ci-dessous propose une classe qui encapsule des tests avec JUnit 3 et une classe qui propose des fonctionnalités équivalentes en JUnit 4.

Exemple avec JUnit 3 :

```
package com.jmdoudoux.test.junit;

import junit.framework.TestCase;

public class PersonneTest extends TestCase {

  private Personne personne;

  public PersonneTest(String name) {
    super(name);
  }

  protected void setUp() throws Exception {
    super.setUp();
    personne = new Personne("nom1","prenom1");
  }

  protected void tearDown() throws Exception {
    super.tearDown();
    personne = null;
  }

  public void testPersonne() {
    assertNotNull("l'instance est créée", personne);
  }

  public void testGetNom() {
    assertEquals("est ce que nom est correct", "nom1", personne.getNom());
  }

  public void testSetNom() {
    personne.setNom("nom2");
    assertEquals("est ce que nom est correct", "nom2", personne.getNom());
  }

  public void testGetPrenom() {
    assertEquals("est ce que prenom est correct", "prenom1", personne.getPrenom());
  }

  public void testSetPrenom() {
    personne.setPrenom("prenom2");
    assertEquals("est ce que prenom est correct", "prenom2", personne.getPrenom());
  }

}
```

Exemple avec JUnit 4 :

```
package com.jmdoudoux.test.junit4;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotNull;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class PersonneTest {

  private Personne personne;

  @Before
  public void initialiser() throws Exception {
    personne = new Personne("nom1","prenom1");
  }

  @After
  public void nettoyer() throws Exception {
    personne = null;
  }
```

```
  @Test
  public void personne() {
    assertNotNull("l'instance est créée", personne);
  }

  @Test
  public void getNom() {
    assertEquals("est ce que nom est correct", "nom1", personne.getNom());
  }

  @Test(expected=IllegalArgumentException.class)
  public void setNom() {
    personne.setNom("nom2");
    assertEquals("est ce que nom est correct", "nom2", personne.getNom());
    personne.setNom(null);
  }

  @Test
  public void getPrenom() {
    assertEquals("est ce que prenom est correct", "prenom1", personne.getPrenom());
  }

  @Test
  public void setPrenom() {
    personne.setPrenom("prenom2");
    assertEquals("est ce que prenom est correct", "prenom2", personne.getPrenom());
  }

}
```

79.6.7. La limitation du temps d'exécution d'un cas de test

JUnit 4 propose une fonctionnalité rudimentaire pour vérifier qu'un cas de tests s'exécute dans un temps maximum donné.

L'attribut timeout de l'annotation @Test attend comme valeur un délai maximum d'exécution exprimé en millisecondes.

Exemple :

```
package com.jmdoudoux.test.junit4;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotNull;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class PersonneTest {

...

  @Test(timeout=100)
  public void compteur() {
    for(long i = 0 ; i < 999999999; i++) { long a = i + 1; }
  }

  public static void main(String[] args) {
    org.junit.runner.JUnitCore.main("com.jmdoudoux.test.junit4.PersonneTest");
  }
}
```

Si le temps d'exécution du cas de tests est supérieur au temps fourni, alors le cas de tests échoue.

Résultat :

```
JUnit version 4.3.1
......E
Time: 0,141
There was 1 failure:
1) compteur(com.jmdoudoux.test.junit4.PersonneTest)
java.lang.Exception: test timed out after 100 milliseconds
        at org.junit.internal.runners.TestMethodRunner.runWithTimeout
                (TestMethodRunner.java:68)
        at org.junit.internal.runners.TestMethodRunner.run
                (TestMethodRunner.java:43)
        at org.junit.internal.runners.TestClassMethodsRunner.invokeTestMethod
                (TestClassMethodsRunner.java:66)
        at org.junit.internal.runners.TestClassMethodsRunner.run
                (TestClassMethodsRunner.java:35)
        at org.junit.internal.runners.TestClassRunner$1.runUnprotected
                (TestClassRunner.java:42)
        at org.junit.internal.runners.BeforeAndAfterRunner.runProtected
                (BeforeAndAfterRunner.java:34)
        at org.junit.internal.runners.TestClassRunner.run(TestClassRunner.java:52)
        at org.junit.internal.runners.CompositeRunner.run(CompositeRunner.java:29)
        at org.junit.runner.JUnitCore.run(JUnitCore.java:130)
        at org.junit.runner.JUnitCore.run(JUnitCore.java:109)
        at org.junit.runner.JUnitCore.run(JUnitCore.java:100)
        at org.junit.runner.JUnitCore.runMain(JUnitCore.java:81)
        at org.junit.runner.JUnitCore.main(JUnitCore.java:44)
        at com.jmdoudoux.test.junit4.PersonneTest.main(PersonneTest.java:58)

FAILURES!!!
Tests run: 6,  Failures: 1<b></b>
```

79.6.8. Les tests paramétrés

Cette section sera développée dans une version future de ce document

79.6.9. La rétro compatibilité

Il est possible d'exécuter des tests JUnit 4 dans une application d'exécution de Tests JUnit 3.

Pour cela, il faut dans la classe de tests, ajouter une méthode suite() qui retourne un objet de type junit.framework.Test. Cette méthode instancie un objet de type JUnit4TestAdapter qui attend comme paramètre de son constructeur la classe de la classe de tests.

Exemple :

```
package com.jmdoudoux.test.junit4;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotNull;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class PersonneTest {

...
```

```
  public static junit.framework.Test suite() {
    return new JUnit4testAdapter(PersonneTest.class);
  }
}
```

L'exécution nécessite tout de même une version 5 ou supérieure de Java.

79.6.10. L'organisation des tests

Il est généralement préférable de n'avoir qu'un seul assert par test car un test ne devrait avoir qu'une seule raison d'échouer.

Exemple :

```
package com.jmdoudoux.test;

public class SecuriteHelper {

  public static boolean isMotDePasseValide(String mdp) {
    boolean resultat = true;

    if (mdp == null) {
      resultat = false;
      throw new IllegalArgumentException("le mot de passe n'est pas renseigne");
    } else {

      if (mdp.length() < 6 || mdp.length() > 15) {
        resultat = false;
      }

      if (!mdp.matches(".*[a-zA-Z]*[0-9]*[a-zA-Z]")) {
        resultat = false;
      }
    }
    return resultat;
  }
}
```

La méthode en exemple permet de valider un mot de passe en contrôlant quelques règles simples :

- lever une exception si l'argument est null (pour tester la levée de l'exception dans le cas de test)
- la taille doit être comprise entre 6 et 15 caractères
- il faut au moins un chiffre entre deux caractères
- le dernier caractère doit être une lettre

Il est possible d'écrire une classe de tests ne possédant qu'un seul cas de test avec plusieurs asserts.

Exemple :

```
package com.jmdoudoux.test;

import static org.junit.Assert.assertFalse;
import static org.junit.Assert.assertTrue;
import static org.junit.Assert.fail;

import org.junit.Test;

public class SecurirteHelperTest {

  @Test
  public void testIsMotDePasseValide() {
    try {
      SecuriteHelper.isMotDePasseValide(null);
```

```
      fail("Absence de la levee de l'exception IllegaleArgumentException");
    } catch (IllegalArgumentException iae) {
      // l'exception est levée
    }

    assertFalse("Le mot de passe est vide",
      SecuriteHelper.isMotDePasseValide(""));
    assertFalse("Le mot de passe est trop court",
      SecuriteHelper.isMotDePasseValide("aaa"));
    assertFalse("Le mot de passe est trop long",
      SecuriteHelper.isMotDePasseValide("aaaaaaaaaaaaaaaaaaaaaaaaaaaaa"));
    assertFalse("Le mot de passe ne contient pas de chiffre",
      SecuriteHelper.isMotDePasseValide("aaaaa"));
    assertFalse("Le mot de passe contient un chiffre en derniere position",
      SecuriteHelper.isMotDePasseValide("aaaaa6"));

    assertTrue("Le mot de passe est valide",
      SecuriteHelper.isMotDePasseValide("aAa6Aa"));
    assertTrue("Le mot de passe est valide",
      SecuriteHelper.isMotDePasseValide("a@aA6aa"));
    assertTrue("Le mot de passe est valide",
      SecuriteHelper.isMotDePasseValide("abc456def"));
  }
```

Il est préférable d'écrire une méthode par cas de test même si cela nécessite l'écriture de plus de code.

Exemple :

```
package com.jmd.test;

import static org.junit.Assert.assertFalse;
import static org.junit.Assert.assertTrue;
import static org.junit.Assert.fail;

import org.junit.Test;

public class SecurirteHelperTest {

  @Test(expected=IllegalArgumentException.class)
  public void testIsMotDePasseValideNull() {
      SecuriteHelper.isMotDePasseValide(null);
  }

  @Test
  public void testIsMotDePasseValideVide() {
    assertFalse("Le mot de passe est vide",
      SecuriteHelper.isMotDePasseValide(""));
  }

  @Test
  public void testIsMotDePasseValideTropCourt() {
    assertFalse("Le mot de passe est trop court",
      SecuriteHelper.isMotDePasseValide("aaa"));
  }

  @Test
  public void testIsMotDePasseValideTropLong() {
    assertFalse("Le mot de passe est trop long",
      SecuriteHelper.isMotDePasseValide("aaaaaaaaaaaaaaaaaaaaaaaaaaaaa"));
  }

  @Test
  public void testIsMotDePasseValideSansChiffre() {
    assertFalse("Le mot de passe ne contient pas de chiffre",
      SecuriteHelper.isMotDePasseValide("aaaaa"));
  }

  @Test
  public void testIsMotDePasseValideChiffreEnDernier() {
    assertFalse("Le mot de passe contient un chiffre en derniere position",
      SecuriteHelper.isMotDePasseValide("aaaaa6"));
```

```
  }

  @Test
  public void testIsMotDePasseValideAvecMinMaj() {
    assertTrue("Le mot de passe est valide",
      SecuriteHelper.isMotDePasseValide("aAa6Aa"));
  }

  @Test
  public void testIsMotDePasseValideAvecArobase() {
    assertTrue("Le mot de passe est valide",
      SecuriteHelper.isMotDePasseValide("a@aA6aa"));
  }

  @Test
  public void testIsMotDePasseValideStandard() {
    assertTrue("Le mot de passe est valide",
      SecuriteHelper.isMotDePasseValide("abc456def"));
  }

}
```

En plus de s'assurer que tous les cas de tests sont exécutés même si il y en a un qui échoue cela permet aussi de connaître plus précisément le nombre de cas de tests exécuté (10 au lieu de 1 dans l'exemple). Les cas de tests sont aussi plus simples donc plus maintenables.

Il est cependant possible d'utiliser plusieurs asserts dans un cas de tests si ceux-ci concernent un même cas fonctionnel.

Pour des cas plus concrets, il est peut être nécessaire d'utiliser des méthodes de type setUp() ou TearDown() au besoin pour réduire la quantité de code nécessaire à la mise en place du contexte d'exécution de chaque cas de tests.

80. Les objets de type Mock

Les doublures d'objets ou les objets de type mock permettent de simuler le comportement d'autres objets. Ils peuvent trouver de nombreuses utilités notamment dans les tests unitaires où ils permettent de tester le code en maitrisant le comportement des dépendances.

Ce chapitre contient plusieurs sections :

- ♦ Les doublures d'objets et les objets de type mock
- ♦ L'utilité des objets de type mock
- ♦ Les tests unitaires et les dépendances
- ♦ L'obligation d'avoir une bonne organisation du code
- ♦ Les frameworks
- ♦ Les outils pour générer des objets mock
- ♦ Les inconvénients des objets de type mock

80.1. Les doublures d'objets et les objets de type mock

En POO, il existe plusieurs types d'objets permettant de simuler le comportement d'un autre objet, généralement appelé doublures :

- dummy (fantôme, bouffon) : objets "vides" qui n'ont pas de fonctionnalités implémentées.
- stub (bouchon) : classes qui renvoient en dur une valeur pour une méthode invoquée
- fake (substitut, simulateur) : classes qui est une implémentation partielle par exemple renvoient toujours les mêmes réponses selon les paramètres fournis
- spy (espion) : classe qui vérifie l'utilisation qui en est faite après l'exécution
- mock (simulacre) : classes qui agissent comme un stub et un spy

Le vocabulaire lié à ces types d'objets est assez confus dans la langue anglaise donc il l'est d'autant plus dans la langue française où il y a une tentative de traduction de ce vocabulaire. Ce chapitre va se concentrer essentiellement sur les objets de type mock.

Un objet de type doublure permet donc de simuler le comportement d'un autre objet concret de façon maitrisée.

L'emploi de doublures est largement utilisé pour les tests unitaires mais elle peut aussi être mise en oeuvre lors des développements pour par exemple remplacer un objet qui n'est pas encore écrit.

L'utilisation des doublures permet aux tests unitaires de se concentrer sur les tests du code de la méthode qui correspond au System Under Test (SUT) sans avoir à se préoccuper des dépendances.

Ainsi, les doublures ont pour rôle de simuler le comportement d'un objet permettant de réaliser les tests de l'objet de façon isolée et répétable.

Un objet de type mock permet de simuler le comportement d'un autre objet concret de façon maitrisée et de vérifier les invocations qui sont faites de cet objet.

Cette double fonctionnalité permet dans un test unitaire de faire des tests sur l'état (state test) et des tests sur le comportement (behavior test).

80.1.1. Les types d'objets mock

Il existe deux grands types d'objets mock :

- statique : ce sont des classes Java écrites ou générées via un outil par le développeur
- dynamique : ils sont mis en oeuvre via un framework

Les objets mock peuvent être codés manuellement ou utiliser un framework qui va permettre de les générer dynamiquement. L'avantage des mocks dynamiques c'est qu'aucune classe implicite n'a besoin d'être écrite.

Les frameworks de mocking peuvent utiliser plusieurs solutions pour mettre en oeuvre des mocks dynamiques :

- proxy : un proxy est un objet qui est utilisé à la place d'un autre objet. Il est alors nécessaire de fournir ce proxy à l'objet qui l'utilise en utilisant un constructeur ou un setter. Ceci nécessite donc qu'un mécanisme d'injection de dépendance soit mis en oeuvre dans la classe à tester (EasyMock, ...)
- instrumentation : un classloader spécifique est utilisé pour dynamiquement charger une classe à la place d'une autre notamment en utilisant la classe java.lang.Instrument de Java 1.5 (jmockit)
- AOP : permet d'invoquer la méthode d'un mock à la place de celle d'une implémentation concrète sans avoir à mettre en oeuvre une interface ni à requérir un mécanisme d'injection de dépendances. L'invocation de la méthode est interceptée et remplacée par l'invocation de la méthode du mock. Ceci ne doit cependant pas être une excuse pour ne pas écrire du code qui mette en oeuvre une bonne conception car en plus d'être testable cela rend le code plus compréhensible, plus maintenable et plus évolutif. (jeasytest, amock, ...)

Avec l'utilisation de proxy, il est indispensable d'avoir un mécanisme d'injection de dépendances permettant de fournir l'implémentation à utiliser. Ceci permet dans le cas des tests unitaires de fournir un objet de type mock qui sera utilisé lors de l'exécution des tests à la place d'une vraie instance de classe dépendante.

Ce mécanisme d'injection de dépendances peut être fourni par un framework (exemple : Spring) ou implémenté manuellement mais dans tous les cas le code à tester doit fournir un mécanisme pour la réaliser.

Il existe plusieurs frameworks de mocking en Java qui permettent de créer dynamiquement des objets de type mock.

80.1.2. Exemple d'utilisation dans les tests unitaires

Dans une application, les classes ont généralement des dépendances entre elles. Ceci est particulièrement vrai dans les applications développées en couches (présentation, service, métier, accès aux données (DAO), ...).

L'idée lors de l'exécution d'un test unitaire est de tester la plus petite unité de code possible, soit la méthode et uniquement le code de la méthode. Hors les classes utilisées dans le code de cette méthode font généralement appel à un ou plusieurs autres objets. Le but n'est pas de tester ces objets qui feront eux même l'objet de tests unitaires mais de tester le code de la méthode : le test unitaire doit concerner uniquement la méthode et ne pas tester les dépendances.

Il faut donc une solution pour s'assurer que les objets dépendants fournissent les réponses désirées à leur invocation. Cette solution repose sur les objets de type simulacre.

Cela suppose que si le code de la méthode fonctionne comme voulu (validée par des tests unitaires) et que les dépendances fonctionnent comme voulu (validées par leurs tests unitaires) alors ils fonctionneront normalement ensembles.

Dans un test unitaire, les classes dépendantes ne doivent pas être testées dans les tests unitaires de la classe. Elles doivent être considérées comme testées, sachant que des tests unitaires qui leur sont dédiés doivent exister. Certaines classes doivent aussi être considérées comme testées : c'est notamment le cas des classes du JRE.

Il est très important que les tests unitaires ne concernent que le code de la méthode en cours de test. Autrement, il est

difficile de trouver un bug qui peut être dans un objet dépendant de niveau -N.

Il est alors nécessaire de simuler le fonctionnement des classes dépendantes.

Le but d'un objet Mock est de remplacer un autre objet en proposant de forcer les valeurs de retour de ses méthodes selon certains paramètres.

Ainsi l'invocation d'un objet de type mock garantie d'avoir les valeurs attendues selon les paramètres fournies.

80.1.3. La mise en oeuvre des objets de type mock

Un des avantages d'utiliser des objets mock, notamment dans les tests unitaires, est qu'il force le code à être écrit ou adapté via des refactoring pour qu'il respecte une conception qui permettre de rendre le code testable.

Généralement, un objet de type mock est une implémentation d'une interface qui se limite généralement à renvoyer des valeurs déterminées en fonction des paramètres reçus. L'interface est parfaitement adaptée puisque l'objet simulé et l'objet mock doivent avoir le même contrat.

Un objet de type mock possède donc la même interface que l'objet qu'il doit simuler, ce qui permet d'utiliser le mock ou une implémentation concrète de façon transparente pour l'objet qui l'invoque.

Les objets mock simulent le comportement d'autres objets mais ils sont aussi capables de vérifier les invocations qui sont faites sur le mock : nombres d'invocations, paramètres fournis, ordre d'invocations, ...

La mise en oeuvre d'un objet de type mock dans les tests unitaires suit généralement plusieurs étapes :

- définir le comportement du mock : méthodes invoquées, paramètres fournis, valeurs de retour ou exception ...
- exécuter le test en invoquant la méthode à tester
- vérifier des résultats du test
- vérifier les invocations du ou des objets de type mock : nombre d'invocations, ordre d'invocations, ...

80.2. L'utilité des objets de type mock

Les objetsde type mock peuvent être utilisés dans différentes circonstances :

- renvoyer des résultats déterminés notamment dans des tests unitaires automatisés
- obtenir un état difficilement reproductible (erreur d'accès réseau, ...)
- éviter d'invoquer des ressources longues à répondre (accès à une base de données, ...)
- invoquer un composant qui n'existe encore pas
- ...

Les objets de type mock sont donc très intéressants pour simuler le comportement de composants invoqués de façon distante (exemple : EJB, services web, RMI, ...) et particulièrement pour tester les cas d'erreurs (problème de communication, défaillance du composant ou du serveur qui gère leur cycle de vie, ...).

80.2.1. L'utilisation dans les tests unitaires

Les tests unitaires automatisés sont une composante très importante du processus de développement et de maintenance d'une application, malgré le fait qu'ils soient fréquemment négligés.

Pour permettre de facilement détecter et corriger d'éventuels bugs dans le code testé, il est nécessaire d'isoler ce code en simulant le comportement de ces dépendances.

L'utilisation des objets mock est une technique particulièrement puissante pour permettre des tests unitaires sur des classes.

Les objets de type mock permettent réellement des tests qui soient unitaires puisque leur résultat est prévisible, si le test échoue, il y a une forte probabilité que l'origine du problème soit dans la méthode en cours de tests. Ceci facilite la résolution du problème puisque celui-ci est isolé uniquement dans le code en cours de tests.

Les objets de type mock permettent de s'assurer que l'échec d'un test n'est pas lié à une de ces dépendances sauf si les données retournées par le ou les objets mock sont erronées vis-à-vis du cas de test en échec.

80.2.2. L'utilisation dans les tests d'intégration

Les objets mock sont particulièrement utiles dans les tests unitaires mais ils sont à éviter dans les tests d'intégration. Le but des tests d'intégration étant de tester les interactions entre les modules et les composants, il n'est pas forcement souhaitable de simuler le comportement de certains d'entre eux.

Pour les tests d'intégration, les objets mock peuvent cependant être utiles dans certaines circonstances :

- pour simuler un module dont le temps de traitement est particulièrement long
- pour simuler un module qui est complexe à initialiser
- pour simuler des cas d'erreur

80.2.3. La simulation de l'appel à des ressources

Les tests unitaires doivent toujours s'exécuter le plus rapidement possible notamment si ceux-ci sont intégrés dans un processus de build automatique. Un test unitaire ne doit donc pas utiliser de ressources externes comme une base de données, des fichiers, des services, ... Les tests avec ces ressources doivent être faits dans les tests d'intégration puisque se sont des dépendances.

80.2.4. La simulation du comportement de composants ayant des résultats variables

Les tests utilisant une fonctionnalité dont le résultat est aléatoire ou fluctuant selon les appels avec le même contexte ne sont pas répétables.

Exemple : une méthode qui convertit le montant d'une monnaie dans une autre. La méthode utilise un service web pour obtenir le cours de la monnaie cible. A chaque exécution du cas de test, le résultat peut varier puisque le cours d'une monnaie fluctu.

Pour permettre d'exécuter correctement les tests d'une méthode qui utilise une telle fonctionnalité, il faut simuler le comportement du service dépendant pour qu'il retourne des valeurs prédéfinies selon le contexte fourni en paramètre. Ainsi pour chaque cas de tests, le service retournera la même valeur rendant ainsi les résultats prédictibles et donc les tests répétables.

Bien sûre ce type de tests pose comme pré-requis que le service fonctionne correctement mais cela est du ressort des développeurs du service qui doivent eux aussi garantir le bon fonctionnement de leur service ... en utilisant des tests unitaires.

80.2.5. La simulation des cas d'erreurs

Les objets de type mock peuvent aussi permettre de facilement tester des cas d'erreurs. Certaines erreurs sont difficiles à reproduire donc à tester, par exemple un problème de communication avec le réseau, d'accès à une ressource, de connexion à un serveur (Base de données, Broker de messages, système de fichiers partagés,...).

Il est possible d'effectuer des opérations manuelles pour réaliser ces tests (débrancher le câble réseau, arrêter un serveur, ...) mais ces opérations sont fastidieuses et peux automatisables.

Il est par contre très facile pour un objet Mock de retourner une exception qui va permettre de simuler et de tester le cas d'erreur correspondant.

80.3. Les tests unitaires et les dépendances

Le principe de limiter les responsabilités d'un objet pour faciliter la réutilisation implique qu'un objet a souvent besoin d'autres objets pour réaliser ses tâches. Ces objets dépendants ont eux aussi des dépendances vers d'autres objets. Ces dépendances forment rapidement un graphe d'objets complexe qui pose rapidement des problèmes pour les tests et surtout qui empêche l'isolation du test de l'objet à tester. Cela devient particulièrement vrai si une ou plusieurs de ces dépendances utilisent des ressources distantes, longues à répondre ou dont les résultats ne sont pas constants.

Le test unitaire d'un objet peut utiliser des objets de type mock pour simuler le comportement de leurs dépendances immédiates.

Seules les dépendances de premier niveau ont besoin d'être remplacées par des objets mock pour tester l'objet. Pour tester l'objet, il est inutile de créer des mocks pour les dépendances de niveau 2 et supérieur.

80.4. L'obligation d'avoir une bonne organisation du code

Un code mal conçu pour être testable est un frein à la rédaction des tests unitaires automatisés car ceux ci seront trop compliqué voir impossible à écrire.

Les tests unitaires et en encore plus l'utilisation d'objets de type mock encourage voir impose l'utilisation d'une conception adaptée du code qui au final le rend non seulement testable mais aussi plus comprehensible et plus maintenable.

L'écriture de tests unitaires impose que le code écrit soit testable, ce qui implique notamment que :

- le code d'une méthode ne doit pas être trop important (amélioration du découpage des fonctionnalités)
- le code ne doit pas être complexe
- le code ne doit pas avoir trop de dépendances
- les dépendances doivent pouvoir être injectées
- ...

80.4.1. Quelques recommandations

Si une classe dépendante est simplement instanciée dans le code de la méthode à tester, il ne sera pas possible de la simuler en la remplaçant par un autre objet. Pour faciliter les tests unitaires qui utilisent des objets mocks, il faut mettre en oeuvre un mécanisme d'injection de dépendances et définir une interface pour chaque objet dépendant.

Les dépendances doivent être décrites via une interface pour permettre facilement de créer des objets de type mocks. Il est possible de créer une instance d'une interface sans avoir à fournir d'implémentations des méthodes. Les objets mocks vont utiliser cette fonctionnalité pour fournir une implémentation qui va simuler le comportement du véritable objet.

L'utilisation de certaines fonctionnalités ou motifs de conception peut entraver, voir rendre compliqué et même impossible le test du code avec des tests unitaires, par exemple :

- il faut éviter le motif de conception singleton
- il ne faut pas utiliser d'initialisation static pour initialiser les propriétés
- ...

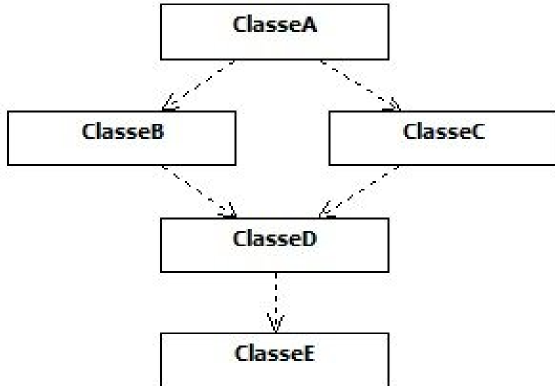
Si l'accès à une ressource est codé de façon statique, cette ressource devra être disponible lors de l'exécution des tests et elle sera sollicitée lors de leur exécution.

Il est généralement préférable d'effectuer un refactoring pour rendre le code testable plutôt que d'écrire des tests unitaires compliqués voir de ne pas les écrire du tout. Au final, le code est amélioré.

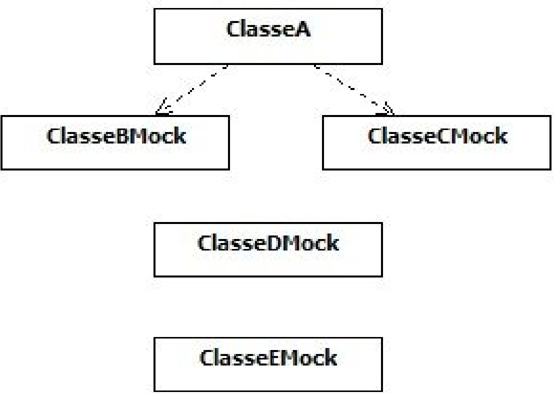
80.4.2. Les dépendances et les tests unitaires

Les objets de type mock permettent de réaliser des tests unitaires dans le contexte d'un système reposant sur des développements orienté objets. Dans un tel contexte, il existe des dépendances plus ou moins nombreuses entre les objets.

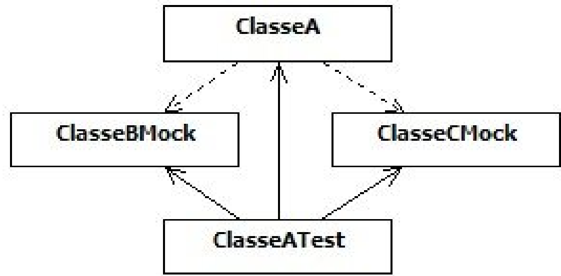
Ainsi la méthode d'une classe, qui est l'unité à tester, dépends généralement d'un ou plusieurs objets dépendant eux aussi d'autres objets. L'invocation de la méthode lors de son test va inévitablement faire appel à ces dépendances : ceci n'est alors plus un test unitaire mais un test d'intégration. De plus, cela complexifie généralement la détermination de l'origine d'un problème et l'induction de difficultés de répétatibilité de l'exécution des tests.



Les objets de type mock permettent de maitriser le fonctionnement des dépendances en simulant de façon prédéterminée leur comportement lors de l'exécution des cas de tests.

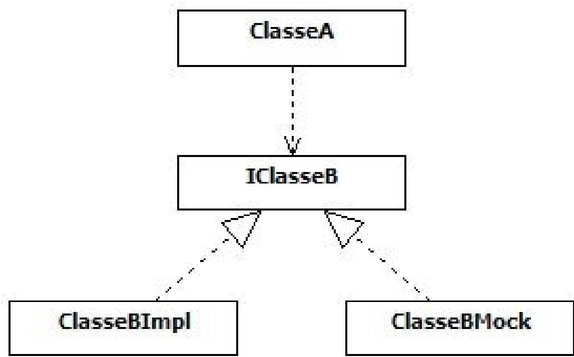


Les objets de type mock sont donc conçus pour répondre au besoin des tests unitaires.



L'entité testée ne doit pas savoir si l'objet utilisé durant les tests est un mock. Pour cela, les dépendances d'une entité testée doivent être décrites avec une interface et l'entité doit utiliser un mécanisme d'injection pour permettre d'utiliser une implémentation de la dépendance en production et d'utiliser un mock pour les tests.

Le fait de décrire les fonctionnalités d'une dépendance via une interface facilite la mise oeuvre d'un objet de type mock.



L'injection de dépendances doit permettre de substituer l'implémentation de la dépendance par un objet de type mock. Si le code à tester instancie en dur la dépendance en utilisant l'opérateur new, il est extrêmement difficile de réaliser la substitution sauf en mettant en oeuvre des fonctionnalités avancées et complexes (via un classloader par exemple).

L'injection peut se faire avec plusieurs solutions, par exemple :

- elle peut se faire directement sur le type de l'interface
- sur une fabrique qui se charge de créer l'instance voir un mixe de ces deux solutions
- définir une méthode protected qui renvoie une instance de l'interface (par défaut celle de l'implémentation). Pour les tests, il suffit alors d'hériter de la classe à tester et de redéfinir ce getter pour renvoyer une instance du mock
- utiliser un framework qui propose une solution d'injection de dépendance (Exemple : Spring)
- ...

80.4.3. Exemple de mise en oeuvre de l'injection de dépendances

Cette section va faire évoluer une classe dont une méthode à tester utilise une dépendance directement instanciée.

L'important pour pouvoir utiliser les mocks c'est que la classe ait été conçue et développée pour être testable. Ceci implique notamment de proposer un mécanisme permettant de pouvoir remplacer une implémentation d'une dépendance par un objet de type mock.

Exemple :

```
public class ClasseA {
  public String maMethode(){
     // début des traitements
    ClasseB classeB = new ClasseB();
    // suite des traitements utilisant classeB
  }
```

```
}
```

Dans l'exemple ci dessus, le test de la méthode maMethode() va être difficile car l'instanciation de la dépendance se fait en dur dans le code. Il ne va pas être facile de remplacer cette instance par celle d'un objet mock. Une solution consiste à proposer une méthode qui se charge de retourne une instance de la classe ClasseB. Il est important que cette méthode puisse t-être redéfinie dans une classe fille.

Exemple :

```
public class ClasseA {
  public String maMethode(){
    // début des traitements
    ClasseB classeB = creerClasseB();
    // suite des traitements utilisant classeB
  }

  protected ClasseB creerClasseB() {
    return new ClasseB();
  }
}
```

Pour faciliter la création d'un objet mock, il est préférable que chaque objet dépendant implémente une interface qui décrive ses fonctionnalités.

Exemple :

```
public class ClasseB implement InterfaceB {

...
}
```

L'objet doit alors utiliser des dépendances du type de leur interface. Il est alors plus facile de créer un objet mock car cela évite de dériver la classe.

Exemple :

```
public class ClasseA {
  public String maMethode(){
    // début des traitements
    InterfaceB classeB = creerClasseB();
    // suite des traitements utilisant classeB
  }

  protected InterfaceB creerClasseB() {
    return new ClasseB();
  }
}
```

Lors de l'écriture du test, il faut dériver la classe à tester et réécrire la méthode qui instancie la dépendance pour qu'elle renvoie une instance de l'objet mock.

Exemple :

```
public class TestClasseA extends TestCase {

  public void testMaMethode() {
    ClasseA classeA = new ClasseA(){
      // reecriture de la méthode pour qu'elle renvoie un mock
      protected InterfaceB creerClasseB() {
        // renvoie une instance du mock de la classe B
        return new MockB();
      }
    }
```

```
    String resultat = classeA.maMethode();
    // évaluation des résultats du cas de test
  }
}
```

Cet exemple permet de facilement mettre en oeuvre le principe d'injection de dépendances dans du code qui n'a rien pour mettre en oeuvre ce type de fonctionnalité proposée par certains frameworks, par exemple Spring.

80.4.4. Limiter l'usage des singletons

Les singletons ne permettent pas facilement de remplacer leur unique instance par un objet de type mock.

80.4.5. Encapsuler les ressources externes

Il faut encapsuler les dépendances vers des ressources externes dans des entités dédiées pour permettre de les injecter et ainsi pour utiliser une implémentation à l'exécution et un objet mock lors des tests.

80.5. Les frameworks

L'écriture d'objets de type mock à la main peut être long et fastidieux et peut contenir comme toute portion de code des bugs. Pour faciliter leur mise en oeuvre, des frameworks ont été créés pour permettre de créer dynamiquement des objets mock de façon fiable.

La plupart des frameworks de mocking permettent de spécifier le comportement que doit avoir l'objet mock :

- les méthodes invoquées : paramètres d'appels et valeur de retour
- l'ordre d'invocations de ces méthodes
- le nombre d'invocations de ces méthodes

Les frameworks de mocking permettent de créer dynamiquement des objets mocks, généralement à partir d'interfaces. Ils proposent fréquemment des fonctionnalités qui vont bien au de là de la simple simulation d'une valeur de retour :

- simulation de cas d'erreur en levant des exceptions
- validation de l'appel de méthodes
- validation de l'ordre de ces appels
- ...

Plusieurs frameworks relatifs aux objets de type mock existent dans le monde Java, notamment :

- EasyMock
- JMockIt
- Mockito
- JMock
- MockRunner

Les différents frameworks vont être utilisés pour mocker les dépendances dans les tests unitaires de la classe ci-dessous qui n'a qu'un rôle purement éducatif.

Exemple :

```
package com.jmdoudoux.test;

public class MonService {
```

```
  protected Calculatrice creerCalculatrice() {
    return new CalculatriceImpl();
  }

  /**
   * Calculer la somme de deux entiers positifs
   *
   * @param val1
   *          la premiere valeur
   * @param val2
   *          la seconde valeur
   * @return la somme des deux arguments ou -1 si un des deux arguments est
   *         negatif
   */
  public long additionner(int val1, int val2) {
    long retour = 0l;
    Calculatrice calculatrice = creerCalculatrice();

    try {
      retour = calculatrice.additionner(val1, val2);
    } catch (IllegalArgumentException iae) {
      retour = -1l;
    }

    return retour;
  }

  /**
   * Calculer la somme de deux premiers parametres et soustraire la valeur du troisième
   * @param val1
   * @param val2
   * @param val3
   * @return le resultat du calcul
   */
  public long calculer(int val1, int val2, int val3) {
    long retour = 0l;
    Calculatrice calculatrice = creerCalculatrice();

    try {
      long somme = calculatrice.additionner(val1, val2);
      retour = calculatrice.soustraire(somme, val3);
    } catch (IllegalArgumentException iae) {
      retour = -1l;
    }

    return retour;
  }
}
```

Cette classe utilise une dépendance dont les fonctionnalités sont décrites dans une interface.

Exemple :

```
package com.jmdoudoux.test;

public interface Calculatrice {

  public long additionner(int val1, int val2);

  public long soustraire(long val1, int val2);

}
```

Exemple :

```
package com.jmdoudoux.test;

public class CalculatriceImpl implements Calculatrice {

  @Override
```

```
  public long additionner(int val1, int val2) {
    if (val1 <  0 || val2 < 0) {
      throw new IllegalArgumentException("La valeur ne peut pas etre negative");
    }
    return val1+val2;
  }

  @Override
  public long soustraire(long val1, int val2) {
    return val1-val2;
  }

}
```

80.5.1. EasyMock

Easy Mock est un framework de mocking open source qui permet de créer et d'utiliser des objets de type mock.

EasyMock est un framework simple (composé uniquement d'une douzaine de classes et interfaces) mais très puissant pour définir et utiliser des objets de type mock.

EasyMock travaille à partir d'interfaces pour créer des objets de type mock mais il propose une extension qui permet de créer des objets de type mock pour des classes.

Les mocks créés par EasyMock peuvent avoir plusieurs utilités allant du simple dummy qui renvoie une valeur au spy qui permet de vérifier le comportement des invocations de méthodes selon les paramètres fournis.

La version utilisée dans cette section est la 2.4. Elle requiert un Java 5 minimum.

Pour l'utiliser, il faut télécharger l'archive sur le site http://easymock.org/ et la décompresser dans un répertoire du système. Il suffit alors d'ajouter le fichier easymock.jar dans le classpath.

La classe principale d'EasyMock est la classe EasyMock : toutes ces méthodes sont statiques. Il est donc possible de faire un import static de cette classe pour pouvoir invoquer ces méthodes sans avoir à les préfixer par le nom de la classe.

Voici un exemple de tests unitaires de la classe MonService qui utilise EasyMock pour simuler le comportement des dépendances.

Exemple :

```
package com.jmdoudoux.test;

import org.easymock.EasyMock;
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;

public class MonServiceTest {

  private Calculatrice mock = null;

  private MonService monService = null;

  @Before
  public void setUp() throws Exception {
    mock = EasyMock.createMock(Calculatrice.class);

    monService = new MonService() {
      @Override
      protected Calculatrice creerCalculatrice() {
        return mock;
      }
    };
  }
```

```
  @Test
  public void testAdditionner() {
    long retour = 0l;

    EasyMock.expect(mock.additionner(1, 2)).andReturn(Long.valueOf(3l));
    EasyMock.replay(mock);

    retour = monService.additionner(1, 2);
    Assert.assertEquals("Valeur retournee est invalide", 3l, retour);
  }

  @Test
  public void testAdditionnerParametreInvalide() {
    long retour = 0l;

    EasyMock.expect(mock.additionner(-1, 2)).andThrow(new IllegalArgumentException());
    EasyMock.replay(mock);

    retour = monService.additionner(-1, 2);
    Assert.assertEquals("Valeur retournee est invalide", -1l, retour);
  }

  @Test
  public void testCalculer() {
    long retour = 0l;

    EasyMock.expect(mock.additionner(20, 30)).andReturn(Long.valueOf(50l));
    EasyMock.expect(mock.soustraire(50, 10)).andReturn(Long.valueOf(40l));
    EasyMock.replay(mock);

    retour = monService.calculer(20, 30, 10);
    Assert.assertEquals("Valeur retournee est invalide", 40l, retour);
  }

}
```

80.5.1.1. La création d'objets mock

Par défaut, EasyMock ne permet que de créer des objets de type mock uniquement pour des interfaces. EasyMock propose une extension pour créer des objets mock à partir d'une classe. Le code à utiliser est similaire hormis qu'il faut importer le package org.easymock.classextension. EasyMock à la place du package org.easymock.EasyMock.

La classe org.easymock.EasyMock permet de créer et utiliser des objets de type mock à partir d'une interface qui précise les fonctionnalités de l'objet à simuler.

La classe org.easymock.classestension.EasyMock permet de créer et utiliser des objets de type mock à partir d'une classe.

La création d'une instance d'un objet de type mock se fait en invoquant la méthode statique createMock() de la classe EasyMock qui attend en paramètre la classe de l'interface

Exemple :

```
    mock = EasyMock.createMock(Calculatrice.class);
```

EasyMock propose trois types de mock :

- mock : mock classique qui vérifie l'invocation des méthodes. Ils sont créés en utilisant la méthode EasyMock.createMock()
- strict mock : mock qui vérifie l'invocation des méthodes ainsi que l'ordre de ces invocations. Ils sont créés en utilisant la méthode EasyMock.createStrictMock()
- nice mock : mock qui renvoie une valeur par défaut lors de l'invocation d'une méthode du mock dont le comportement n'a pas été précisé. Ils sont créés en utilisant la méthode EasyMock.createNiceMock()

80.5.1.2. La définition du comportement des objets mock

La définition du comportement des mocks se fait sous la forme enregistrer/rejouer.

Pour préciser le comportement d'une méthode d'un mock qui renvoie une valeur, il faut :

- utiliser la méthode statique EasyMock.expect() pour l'invoquer en lui précisant ses paramètres
- utiliser la méthode andReturn() de l'objet de type IExpectationSetters retourné par l'appel précédent pour préciser la valeur de retour

La méthode expect() permet de préciser le comportement attendu en retour de l'invocation d'une méthode.

L'avantage de cette approche qui nécessite l'invocation de la méthode dont le comportement est à simuler est qu'elle permet d'utiliser le code completion et le refactoring de l'IDE utilisé.

Exemple :

```
EasyMock.expect(mock.additionner(1, 2)).andReturn(Long.valueOf(3l));
```

EasyMock propose de simuler la levée d'une exception dans la définition du comportement d'une méthode en utilisant la méthode andThrow() qui attend en paramètre l'instance de l'exception à lever.

Exemple :

```
EasyMock.expect(mock.additionner(-1, 2)).andThrow(new IllegalArgumentException());
```

Ceci est particulièrement utile pour tester des cas d'erreurs difficiles à automatiser comme une coupure réseau par exemple. Tous les types d'exceptions peuvent être simulés (checked, runtime ou error).

Pour préciser le comportement d'une méthode qui ne retourne aucune valeur (void), il faut simplement invoquer la méthode sur l'instance du mock sans utiliser la méthode expect(). EasyMock va simplement enregistrer le comportement.

Ainsi, si le résultat de l'invocation de la méthode ne renvoie aucune valeur ni ne lève aucune exception, il ne faut pas utiliser la méthode expect() mais simplement invoquer la méthode sur l'objet de type mock.

Certaines méthodes permettent de préciser le nombre d'invocations d'une méthode :

- times(n,m) : la méthode devra être invoquée entre n et m fois
- atLeastOnce() : la méthode devra être invoquée au moins une fois
- anyTimes() : la méthode peut être invoquée un nombre indéfini de fois

EasyMock permet aussi facilement de définir des comportements différents lors de plusieurs invocations de la même méthode car les appels aux méthodes andReturn(), and Throw() et times() peuvent être chaînées.

Exemple :

```
EasyMock.expect(mock.additionner(-1, 2))
  .andReturn(Long.valueOf(3l))
  .andThrow(new IllegalArgumentException());
```

Le comportement attendu est ainsi enregistré par l'objet mock jusqu'à l'invocation de la méthode replay().

80.5.1.3. L'initialisation des objets mock

Une fois tous les comportements attendus du ou des mocks définis, il faut invoquer la méthode replay() sur l'objet de type Control.

Exemple :

```
    EasyMock.replay(mock);
```

Si la méthode statique replay() de la classe EasyMock n'est pas invoquée et que des comportements de l'objet mock sont définis alors une exception de type IllegalStateException est levée lors de l'utilisation des objets de type mock.

Exemple :

```
    java.lang.IllegalStateException: missing behavior definition
      for the preceeding method call additionner(20, 30)
        at org.easymock.internal.MockInvocationHandler.invoke(MockInvocationHandler.java:30)
        at org.easymock.internal.ObjectMethodsFilter.invoke(ObjectMethodsFilter.java:61)
        at $Proxy5.soustraire(Unknown Source)
        at com.jmdoudoux.test.MonService.calculer(MonService.java:45)
        at com.jmdoudoux.test.MonServiceTest.testCalculer(MonServiceTest.java:56)
...
```

Le nom de la méthode replay() peut être source de confusions : en fait, elle ne rejoue pas le comportement de l'objet mock mais elle réinitialise son état interne pour permettre de rejouer le comportement lors des futures invocations des méthodes.

L'appel à la méthode replay() permet d'initialiser l'objet mock dans le mode rejouer lui permettant ainsi de reproduire le comportement défini à l'invocation d'une méthode du mock et d'enregistrer ces invocations.

80.5.1.4. La vérification des invocations des objets mock

EasyMock n'est pas utile que pour fournir des réponses déterminées pour des invocations données : il permet aussi de vérifier les paramètres fournis et l'ordre d'invocations des méthodes.

Une exception est levée si les paramètres utilisés lors de l'invocation ne correspondent par ceux définis dans l'objet de type mock.

Résultat :

```
java.lang.AssertionError:
  Unexpected method call additionner(2, 2):
    additionner(1, 2): expected: 1, actual: 0
        at org.easymock.internal.MockInvocationHandler.invoke(MockInvocationHandler.java:32)
        at org.easymock.internal.ObjectMethodsFilter.invoke(ObjectMethodsFilter.java:61)
        at $Proxy5.additionner(Unknown Source)
        at com.jmdoudoux.test.MonService.additionner(MonService.java:24)
        at com.jmdoudoux.test.MonServiceTest.testAdditionner(MonServiceTest.java:33)
```

Cette exception est levée directement par EasyMock et ne nécessite donc aucune assertion implicite.

C'est le mode de fonctionnement pas défaut d'EasyMock. Cependant, parfois ce mode de fonctionnement est trop strict.

Pour définir qu'un comportement ne requiert pas une valeur précise comme paramètre, il faut utiliser la méthode de la classe EasyMock correspondant au type attendu pour préciser la valeur d'un paramètre : anyObject(), anyInt(), anyShort(), anyByte(), anyLong(), anyFloat(), anyDouble() et anyBoolean().

La méthode EasyMock.notNull() permet de préciser qu'une valeur d'un paramètre doit être non null sans fournir plus de précision sur la valeur.

La méthode EasyMock.matches() permet de préciser qu'une valeur d'un paramètre correspond à un certain motif sous la forme d'une expression régulière.

La méthode EasyMock.find() permet de préciser qu'une valeur d'un paramètre doit contenir un sous ensemble correspondant à un certain motif sous la forme d'une expression régulière.

La méthode EasyMock.lt() permet de préciser qu'une valeur d'un paramètre numérique doit être inférieure à la valeur fournie en paramètre. La méthode EasyMock.gt() permet de préciser qu'une valeur d'un paramètre numérique doit être supérieure à la valeur fournie en paramètre.

80.5.1.5. La vérification de l'ordre d'invocations des mocks

Il peut être important de vouloir vérifier l'ordre d'invocations des méthodes d'un objet mock. Attention cependant, ceci induit un couplage entre l'objet et son mock.

EasyMock ne se contente pas de vérifier les valeurs des paramètres de méthodes du mock invoquées et retourner une valeur, il permet aussi de vérifier l'ordre d'invocations des méthodes et de vérifier que seules les méthodes définies dans le comportement sont invoquées. Cette vérification n'est pas faite pas défaut : pour l'activer, il faut utiliser la méthode EasyMock.verify() une fois toutes les invocations des méthodes du mock réalisées.

Exemple :

```
  @Test
  public void testCalculer() {
    long retour = 0l;

    EasyMock.expect(mock.additionner(20, 30)).andReturn(Long.valueOf(50l));
    EasyMock.expect(mock.soustraire(50, 10)).andReturn(Long.valueOf(40l));
    EasyMock.expect(mock.additionner(40, 60)).andReturn(Long.valueOf(100l));
    EasyMock.replay(mock);

    retour = monService.calculer(20, 30, 10);
    Assert.assertEquals("Valeur retournee est invalide", 40l, retour);
  }
```

Ce test s'exécute sans problème.

Exemple :

```
  @Test
  public void testCalculer() {
    long retour = 0l;

    EasyMock.expect(mock.additionner(20, 30)).andReturn(Long.valueOf(50l));
    EasyMock.expect(mock.soustraire(50, 10)).andReturn(Long.valueOf(40l));
    EasyMock.expect(mock.additionner(40, 60)).andReturn(Long.valueOf(100l));
    EasyMock.replay(mock);

    retour = monService.calculer(20, 30, 10);
    Assert.assertEquals("Valeur retournee est invalide", 40l, retour);
    EasyMock.verify(mock);
  }
```

Ce test échoue car lors de la vérification, le comportement précise une seconde invocation de la méthode additionner() qui n'est pas réalisée dans les traitements.

Une exception est alors levée par EasyMock pour signaler la différence entre le comportement défini et les traitements invoqués.

Exemple :

```
java.lang.AssertionError:
  Expectation failure on verify:
    additionner(40, 60): expected: 1, actual: 0
        at org.easymock.internal.MocksControl.verify(MocksControl.java:101)
        at org.easymock.EasyMock.verify(EasyMock.java:1570)
        at com.jmdoudoux.test.MonServiceTest.testCalculer(MonServiceTest.java:59)
```

Ceci peut être particulièrement utile dans certaines circonstances.

Le fonctionnement de la méthode verify() dépend du mode de création de l'objet de type mock. EasyMock propose trois modes de création :

- normal : toutes les méthodes doivent être invoquées avec les arguments fournis sans tenir compte de leur ordre d'invocation. L'appel à une méthode non définie dans le comportement ou avec des paramètres différents fait échouer le test. La création d'un objet mock dans ce mode se fait avec la méthode EasyMock.createMock()
- strict : toutes les méthodes doivent être invoquées avec les arguments fournis en tenant compte de leur ordre d'invocation. L'appel à une méthode non définie dans le comportement ou avec des paramètres différents ou dans un ordre différent fait échouer le test. La création d'un objet mock dans ce mode se fait avec la méthode EasyMock.createStrictMock()
- nice : toutes les méthodes doivent être invoquées avec les arguments fournis sans tenir compte de leur ordre d'invocation. L'appel à une méthode non définie dans le comportement ou avec des paramètres différents renvoie une valeur par défaut selon le type de données retourné. La création d'un objet mock dans ce mode se fait avec la méthode EasyMock.createNiceMock()

80.5.1.6. La gestion de plusieurs objets de type mock

Un objet de type IMocksControl permet de coupler plusieurs objets de type mock. Ce couplage permet de maintenir des relations sur l'ordre du comportement des différents objets mock.

La mise en oeuvre d'EasyMock avec un control nécessite plusieurs instances :

- un objet de type IMocksControl
- plusieurs instances d'objets de type mock

L'interface IMocksControl propose des méthodes similaires pour mettre en oeuvre les mocks notamment : createMock(), replay() et verify().

La méthode checkOrder() permet de préciser si l'ordre d'invocations des méthodes des mocks doit être vérifié ou non.

La méthode reset() permet de supprimer tous les comportements définis dans les mocks du control.

La méthode resetToNice() permet de supprimer tous les comportements définis dans les mocks du control et de basculer les mock en mode nice.

La méthode resetToStrict() permet de supprimer tous les comportements définis dans les mocks du control et de basculer les mock en mode strict.

La méthode resetToDefault() permet de supprimer tous les comportements définis dans les mocks du control et de basculer les mock en mode par défaut.

Toutes ces méthodes agissent sur les mocks définis dans le control.

80.5.1.7. Un exemple complexe

L'exemple de cette section va mocker le comportement d'une classe de type DAO en proposant notamment d'utiliser des objets de type mock pour les classes Connection, PreparedStatement et ResultSet.

Ainsi, il sera possible de tester unitairement la méthode du DAO sans avoir faire appel à la base de données.

La suite de cette section sera développée dans une version future de ce document

80.5.2. Mockito

80.5.3. JMock

La suite de cette section sera développée dans une version future de ce document

80.5.4. MockRunner

Certaines classes de l'API standard sont particulièrement complexe à mocker.

MockRunner propose un ensemble de mocks pour certaines de ces classes.

La suite de cette section sera développée dans une version future de ce document

80.6. Les outils pour générer des objets mock

Plusieurs outils permettent de générer des objets de type mock, notamment :

- MockObjects
- MockMaker
- ...

La suite de cette section sera développée dans une version future de ce document

80.6.1. MockObjects

80.6.2. MockMaker

80.7. Les inconvénients des objets de type mock

La génération d'objets mock n'est pas toujours pratique car cela permet de créer ses objets mais ceux-ci doivent être maintenus au fur et à mesure des évolutions du code à tests.

Il est préférable d'utiliser un framework qui va créer dynamiquement les objets mock. L'inconvénient c'est que le code du test unitaire devient plus important, donc plus complexe donc plus difficile à maintenir.

L'utilisation d'objets de type mock peut coupler les tests unitaires avec l'implémentation des dépendances utilisées. La plupart des frameworks permettent de préciser et de vérifier l'ordre et le nombre d'appels des méthodes mockées qui sont invoquées lors des tests. Si un refactoring est appliqué sur ces méthodes changeant leur ordre d'invocation, le test devra être adapté en conséquence.

La mise en oeuvre d'objets de type mock doit tenir des limites de leur utilisation. Par exemple, elle masque complète les problèmes d'interactions entre des dépendances. C'est pour cette raison que les tests unitaires sont nécessaires mais pas suffisant. Il peut aussi être intéressant de ne pas mocker systématiquement toutes les dépendances.

81. Des bibliothèques open source

81.1. JFreeChart

JFreeChart est une bibliothèque open source qui permet de d'afficher des données statistiques sous la forme de graphiques. Elle possède plusieurs formats dont le camembert, les barres ou les lignes et propose de nombreuses options de configuration pour personnaliser le rendu des graphiques. Elle peut s'utiliser dans des applications standalone ou des applications web et permet également d'exporter le graphique sous la forme d'une image.

http://www.jfree.org/jfreechart/

La version utilisée dans cette section est la 0.9.18.

Pour l'utiliser, il faut télécharger le fichier jfreechart-0.9.18.zip et le décompresser. Son utilisation nécessite l'ajout dans le classpath des fichiers jfreechart-0.9.18.zip et des fichiers .jar présents dans le répertoire lib décompressé.

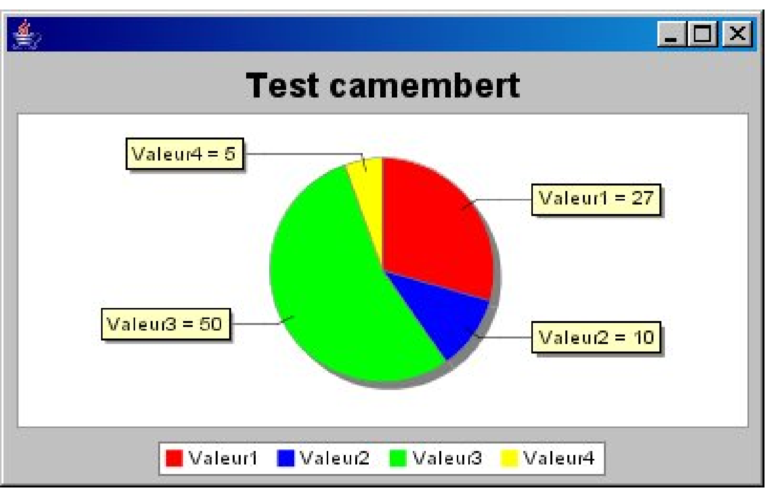
Les données utilisées dans le graphique sont encapsulées dans un objet de type Dataset. Il existe plusieurs sous type de cette classe en fonction du type de graphique souhaité.

Un objet de type JFreechart encapsule le graphique. Une instance d'un tel objet est obtenue en utilisant une des méthodes de la classe ChartFactory.

Exemple : Un exemple avec un graphique en forme de camembert

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import org.jfree.chart.*;
import org.jfree.chart.plot.*;
import org.jfree.data.*;

public class TestPieChart extends JFrame {
  private JPanel pnl;

  public TestPieChart() {
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        dispose();
        System.exit(0);
      }
    });
    pnl = new JPanel(new BorderLayout());
    setContentPane(pnl);
    setSize(400, 250);

    DefaultPieDataset pieDataset = new DefaultPieDataset();
    pieDataset.setValue("Valeur1", new Integer(27));
    pieDataset.setValue("Valeur2", new Integer(10));
    pieDataset.setValue("Valeur3", new Integer(50));
    pieDataset.setValue("Valeur4", new Integer(5));

    JFreeChart pieChart = ChartFactory.createPieChart("Test camembert",
      pieDataset, true, true, true);
    ChartPanel cPanel = new ChartPanel(pieChart);
    pnl.add(cPanel);
```

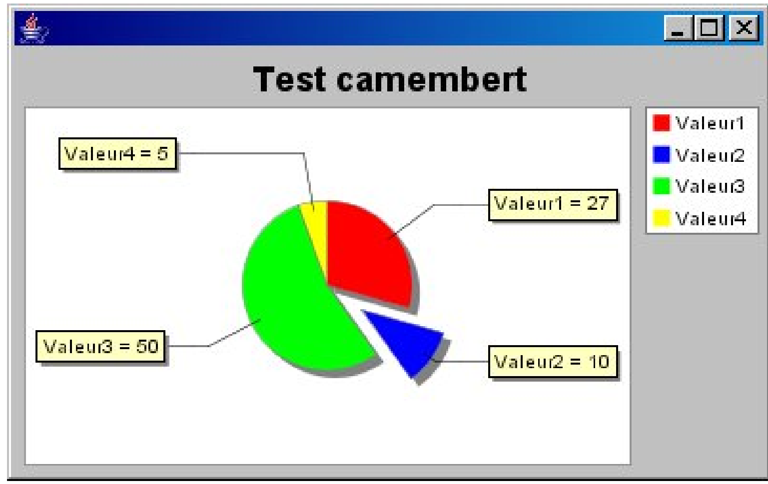
```
  }

  public static void main(String args[]) {
    TestPieChart tpc = new TestPieChart();
    tpc.setVisible(true);
  }
}
```



Pour chaque graphique, il existe de nombreuses possibilités de configuration.

Exemple :

```
...
    JFreeChart pieChart = ChartFactory.createPieChart("Test camembert",
      pieDataset, true, true, true);
    PiePlot piePlot = (PiePlot) pieChart.getPlot();
    piePlot.setExplodePercent(1, 0.5);
    Legend legend = pieChart.getLegend();
    legend.setAnchor(Legend.EAST_NORTHEAST);
    ChartPanel cPanel = new ChartPanel(pieChart);
...
```



Il est très facile d'exporter le graphique dans un flux.

Exemple : enregistrement du graphique dans un fichier

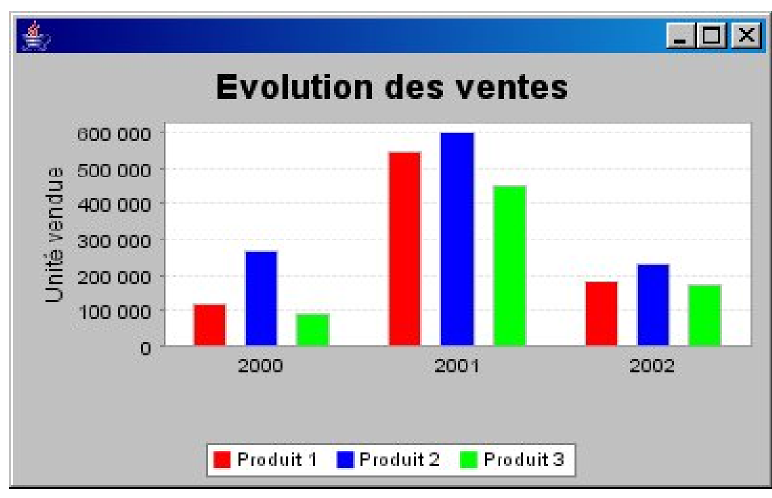
```
...
    File fichier = new File("image.png");
    try {
      ChartUtilities.saveChartAsPNG(fichier, pieChart, 400, 250);
    } catch (IOException e) {
```

```
        e.printStackTrace();
      }
...
```

JFreeChart propose aussi plusieurs autres types de graphiques dont les graphiques en forme de barres.

Exemple : un graphique sous formes de barres

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import org.jfree.chart.*;
import org.jfree.chart.plot.*;
import org.jfree.data.*;

public class TestBarChart extends JFrame {
  private JPanel pnl;

  public TestBarChart() {
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        dispose();
        System.exit(0);
      }
    });
    pnl = new JPanel(new BorderLayout());
    setContentPane(pnl);
    setSize(400, 250);

    DefaultCategoryDataset dataset = new DefaultCategoryDataset();
    dataset.addValue(120000.0, "Produit 1", "2000");
    dataset.addValue(550000.0, "Produit 1", "2001");
    dataset.addValue(180000.0, "Produit 1", "2002");
    dataset.addValue(270000.0, "Produit 2", "2000");
    dataset.addValue(600000.0, "Produit 2", "2001");
    dataset.addValue(230000.0, "Produit 2", "2002");
    dataset.addValue(90000.0, "Produit 3", "2000");
    dataset.addValue(450000.0, "Produit 3", "2001");
    dataset.addValue(170000.0, "Produit 3", "2002");

    JFreeChart barChart = ChartFactory.createBarChart("Evolution des ventes", "",
      "Unité vendue", dataset, PlotOrientation.VERTICAL, true, true, false);
    ChartPanel cPanel = new ChartPanel(barChart);
    pnl.add(cPanel);
  }

  public static void main(String[] args) {
    TestBarChart tbc = new TestBarChart();
    tbc.setVisible(true);
  }
}
```



JFreechart peut aussi être mis en oeuvre dans une application web, le plus pratique étant d'utiliser une servlet qui renvoie dans la réponse une image générée par JfreeChart.

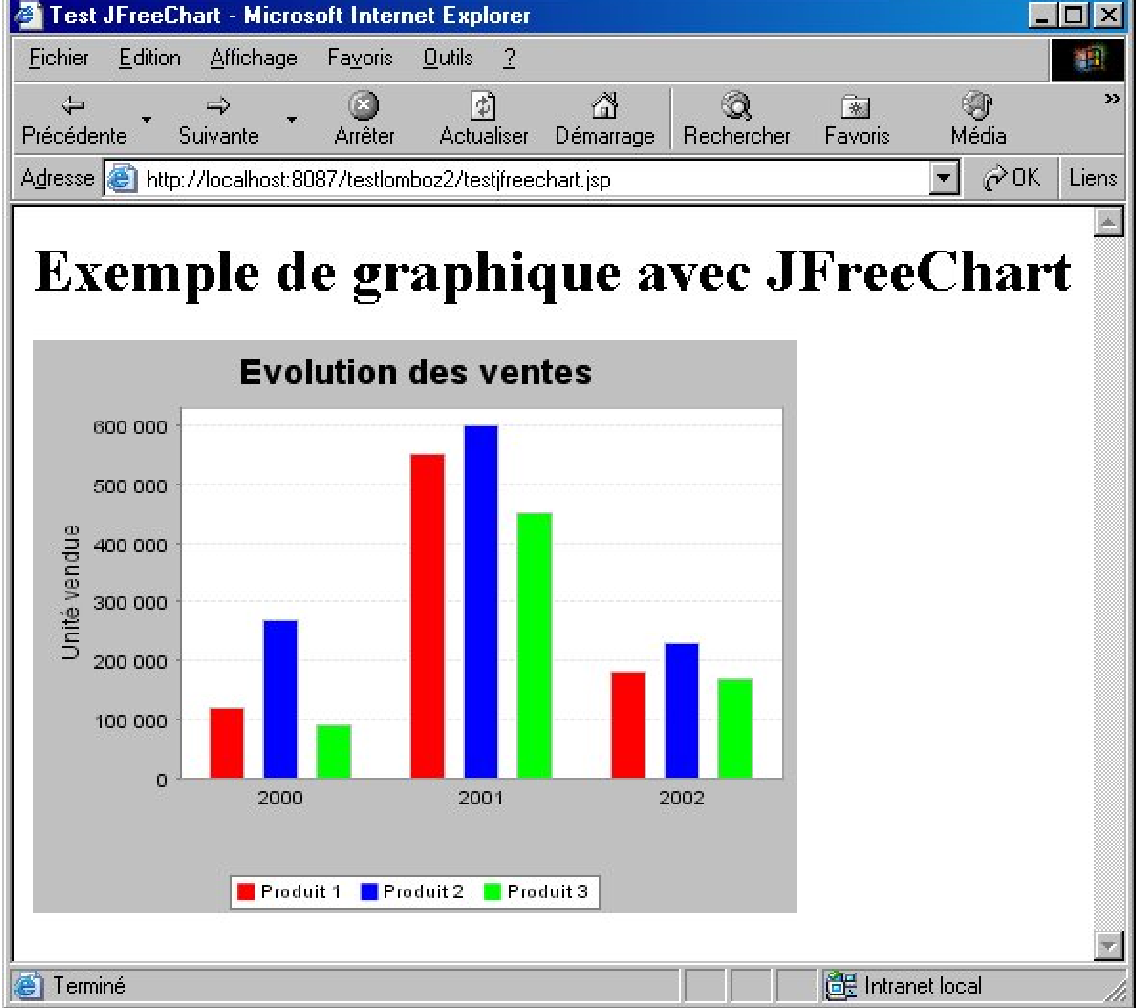
Exemple : JSP qui affiche le graphique

```
<%@ page language="java" %>
<!DOCTYPE HTML PUBLIC "-//w3c//dtd html 4.0 transitional//en">
<html>
<head>
<title>Test JFreeChart</title>
</head>
<body bgcolor="#FFFFFF">
<H1>Exemple de graphique avec JFreeChart</h1>
<img src="/testlomboz2/ServletBarChart">
</body>
</html>
```

Dans l'exemple précédent, l'image contenant le graphique est générée par une servlet.

Exemple : servlet qui génèrère l'image

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import org.jfree.chart.*;
import org.jfree.chart.plot.*;
import org.jfree.data.*;

public class ServletBarChart extends HttpServlet {

  protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {

    DefaultCategoryDataset dataset = new DefaultCategoryDataset();
    dataset.addValue(120000.0, "Produit 1", "2000");
    dataset.addValue(550000.0, "Produit 1", "2001");
    dataset.addValue(180000.0, "Produit 1", "2002");
    dataset.addValue(270000.0, "Produit 2", "2000");
    dataset.addValue(600000.0, "Produit 2", "2001");
    dataset.addValue(230000.0, "Produit 2", "2002");
    dataset.addValue(90000.0, "Produit 3", "2000");
    dataset.addValue(450000.0, "Produit 3", "2001");
    dataset.addValue(170000.0, "Produit 3", "2002");

    JFreeChart barChart = ChartFactory.createBarChart("Evolution des ventes", "",
      "Unité vendue", dataset, PlotOrientation.VERTICAL, true, true, false);
    OutputStream out = response.getOutputStream();
    response.setContentType("image/png");
    ChartUtilities.writeChartAsPNG(out, barChart, 400, 300);
  }
}
```



Cette section n'a proposé qu'une introduction à JFreeChart en proposant quelques exemples très simple sur les nombreuses possibilités de cette puissante bibliothèque.

81.2. Beanshell

Beanshell est un interpréteur de scripts qu'il est possible d'intégrer dans une application.

http://www.beanshell.org/

81.3. Jakarta Commons

81.4. Jakarta ORO

81.5. Joda Time

81.6. Quartz

81.7. JGoodies

81.8. Apache Lucene

82. La génération de documents

Il est fréquent qu'une application de gestion doive produire des documents dans différents formats. Ce chapitre présente plusieurs solutions open source pour permettre la génération de documents notamment au format PDF et Excel.

Ce chapitre contient plusieurs sections :

♦ Apache POI
♦ iText

82.1. Apache POI

POI est l'acronyme de Poor Obfuscation Implementation. C'est un projet open source, sous licence Apache V2, du groupe Apache dont le but est de permettre la manipulation de fichiers de la suite bureautique Office de Microsoft dans des applications Java sans utiliser Office.

L'implémentation de POI est intégralement réalisée en pure Java.

La manipulation ne peut se faire que sur des documents reposant sur le format Microsoft OLE2 (Object Linking and Embedding) Compound Document ce qui inclus les documents de la suite Office mais aussi les applications qui utilisent les ensembles de propriétés MFC pour sérialiser leurs documents.

Ce projet contient plusieurs composants :

- POIFS (Poor Obfuscation Implementation File System) : manipulation de fichiers utilisant le format Microsoft OLE 2 Compound Document
- HSSF (Horrible SpreadSheet Format) : manipulation des fichiers Excel (XLS) en lecture et écriture.
- HWPF (Horrible Word Processor Format) : manipulation de fichiers Word en lecture et certaines fonctionnalités en écriture.
- HPSF (Horrible Slide Layout Format) : manipulation de fichiers PowerPoint en lecture et écriture pour certains fonctionnalités mais pas toutes
- HDGF : lecture et uniquement extraction de texte de fichiers Visio
- HPSF : API pour manipuler les propriétés d'un fichier au format OLE 2 en lecture et en écriture

La version 3.0.1 a été diffusée en juillet 2007.

La version 3.1 a été diffusée fin juin 2008.

La version 3.5 en cours de développement devrait apporter le support des formats Office Open XML proposés depuis la version 2007 d'Office.

Ce projet est particulièrement intéressant car il permet la manipulation de documents au format Office sans que celui-ci soit installé et même sur des systèmes d'exploitation non Microsoft Windows.

Le site officiel du projet est à l'url http://poi.apache.org/

La version utilisée dans cette section est la 3.1

Il faut télécharger l'archive contenant la version binaire de POI à l'url :

http://www.apache.org/dyn/closer.cgi/jakarta/poi/release/

Il faut ensuite décompresser l'archive poi-bin-3.1-FINAL-20080629.zip obtenue dans un répertoire du système.

Pour utiliser PIO, il suffit d'ajouter le fichier poi-3.1-FINAL-20080629.jar au classpath de l'application.

82.1.1. POI-HSSF

HSSF permet la manipulation de document Excel de la version 97 à la version 2007 uniquement pour le format OLE2 (fichier avec l'extension .xls). Le format OOXML d'Excel 2007 n'est pas encours supporté (fichier avec l'extension .xlsx)

HSSF est une solution riche en fonctionnalités et fiable pour la manipulation de documents Excel en Java.

Un document Excel est composé de plusieurs éléments : un Workbook qui contient un ou plusieurs WorkSheets, constitué chacun de Rows composés lui-même de Cells.

Les classes principales de l'API HSSF proposent d'encapsuler chacun de ces éléments.

HSSF propose deux API pour manipuler un document Excel :

- user API : API la plus riche qui permet la lecture et l'écriture mais qui consomme beaucoup de ressources car le document est intégralement représenté dans un graphe d'objets (le pendant pour le traitement de documents XML pourrait être DOM). Les classes de cette API sont regroupées dans le package org.apache.poi.hssf.usermodel
- event API : API pour la lecture uniquement qui consomme peut de ressources (le pendant pour le traitement de documents XML pourrait être SAX). Les classes de cette API sont regroupées dans le package org.apache.poi.hssf.eventmodel et org.apache.poi.hssf.eventusermodel

La liste des packages de HSSF comprend notamment :

Package	Rôle
org.apache.poi.hssf.eventmodel	Classes pour gérer les événements émis lors de la lecture d'un document
org.apache.poi.hssf.eventusermodel	Classes pour lire un document
org.apache.poi.hssf.extractor	Classes pour extraire le texte d'un document
org.apache.poi.hssf.record.formula	Classes pour le support des formules dans les cellules
org.apache.poi.hssf.usermodel	Classes pour la manipulation de documents
org.apache.poi.hssf.util	Utilitaires pour faciliter la mise en oeuvre de certaines fonctionnalités

82.1.1.1. L'API de type usermodel

L'API de HSSF permet de créer, lire et modifier les documents Excel. Pour cela, elle contient de nombreuses classes dont les principales sont :

- POIFSFileSystem : classe qui permet d'accéder à un document existant
- HSSFWorkbook : classe qui encapsule un document
- HSSFSheet : classe qui encapsule une feuille d'un document
- HSSFRow : classe qui encapsule une ligne d'une feuille
- HSSFCell : classe qui encapsule une cellule d'une ligne

Cette API est riche en fonctionnalités mais elle consomme beaucoup de ressources notamment pour des gros de fichiers car ceux-ci sont intégralement représentés en mémoire dans une arborescence d'objets.

Parmi les nombreuses fonctionnalités proposées par cette API, il y a :

- lecture et écriture de document
- création et modification des différentes entités qui composent un document (document, feuille, ligne, cellule, ...)
- support de fonctionnalités avancées sur la feuille : sélection, zoom, support des panneaux, ...
- support des types de données d'une cellule (numérique et date, chaine de caractère, formule)
- formatage des cellules (alignement, police, couleur, bordures, formats de données proposés en standard ou personnalisés,
- fonctionnalités avancées sur les cellules : taille, taille optimale, fusion, commentaires, ...
- paramètre d'impression d'une page (sélection de la zone d'impression, faire tenir sur une page, bas de page, ...)
- support graphique : dessin de primitives, d'images, ...

Seules quelques unes de ces fonctionnalités sont détaillées dans les sections suivantes. Consultez la documentation de l'API pour obtenir des détails sur la mise des autres fonctionnalités.

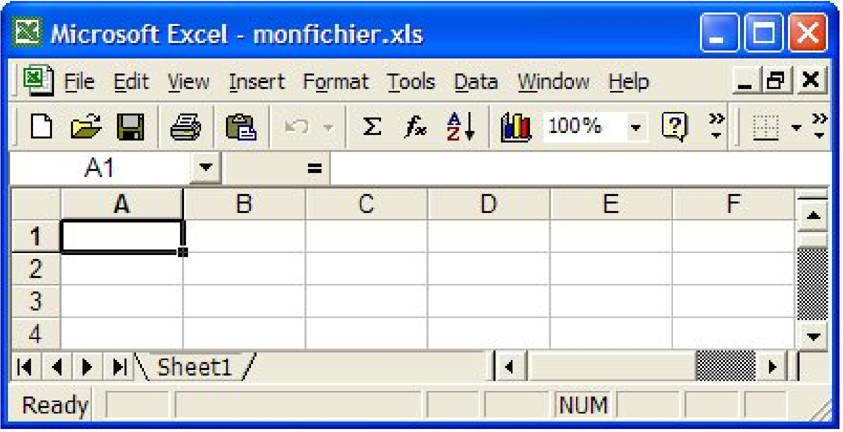
82.1.1.1.1. La création d'un nouveau document

Il suffit d'instancier un objet de type HSSFWorkBook et d'invoquer sa méthode write() pour créer le fichier.

Exemple :

```
package com.jmdoudoux.test.poi;

import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

import org.apache.poi.hssf.usermodel.HSSFWorkbook;

public class TestPOI1 {

  public static void main(
      String[] args) {

    HSSFWorkbook wb = new HSSFWorkbook();
    FileOutputStream fileOut;
    try {
      fileOut = new FileOutputStream("monfichier.xls");
      wb.write(fileOut);
      fileOut.close();
    } catch (FileNotFoundException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

A l'exécution de cet exemple, un document Excel vierge est créé.

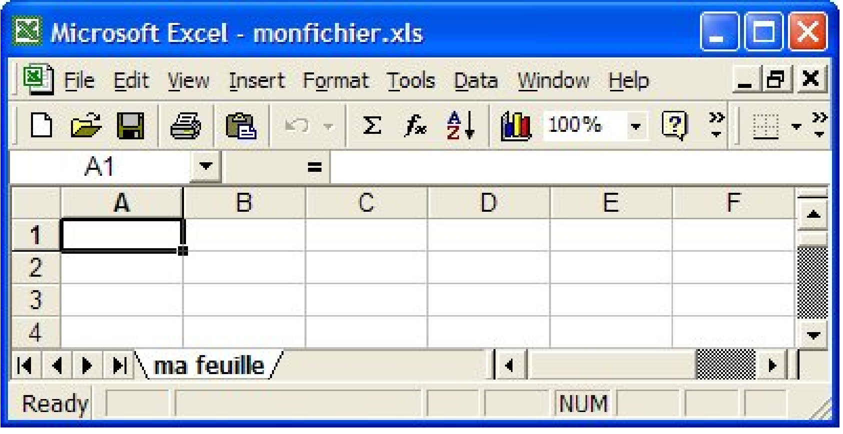


82.1.1.1.2. La création d'une nouvelle feuille

Une feuille est encapsulée dans la classe HSSFSheet. Pour créer une nouvelle feuille dans un document, il faut invoquer la méthode createSheet() de la classe HSSFWorkBook.

Exemple :

```
package com.jmdoudoux.test.poi;

import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

import org.apache.poi.hssf.usermodel.HSSFSheet;
import org.apache.poi.hssf.usermodel.HSSFWorkbook;

public class TestPOI2 {

  public static void main(
      String[] args) {

    HSSFWorkbook wb = new HSSFWorkbook();
    HSSFSheet sheet = wb.createSheet("ma feuille");

    FileOutputStream fileOut;
    try {
      fileOut = new FileOutputStream("monfichier.xls");
      wb.write(fileOut);
      fileOut.close();
    } catch (FileNotFoundException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```



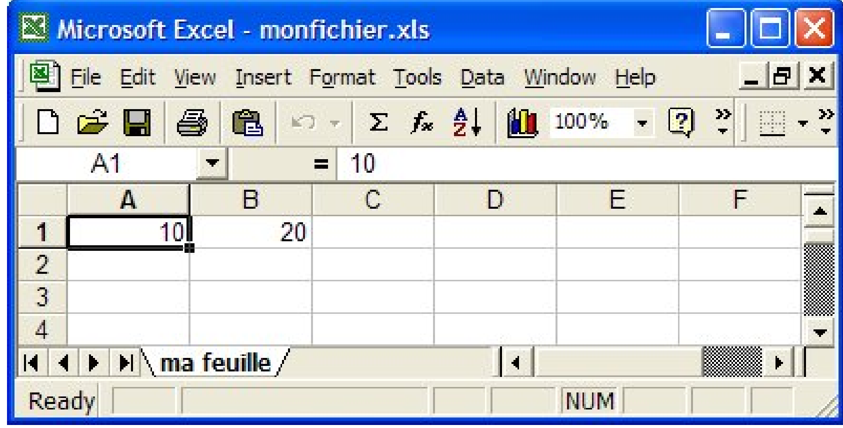
82.1.1.1.3. La création d'une nouvelle cellule

Une cellule d'une feuille est contenue dans une ligne qui est encapsulée dans un objet de type HSSFRow. Pour instancier un objet de ce type, il faut invoquer la méthode createRow() de la classe HSSFSheet. Cette méthode attend en paramètre le numéro de la ligne concernée sous la forme d'un entier de type int sachant que la première ligne possède l'index 0.

Une cellule est encapsulée dans la classe HSSFCell. Pour instancier un objet de ce type, il faut invoquer la méthode createCell() de la classe HSSFRow. Cette méthode attend en paramètre le numéro de la ligne concernée sous la forme d'un entier de type short sachant que la première cellule possède l'index 0.

Exemple :

```
package com.jmdoudoux.test.poi;

import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

import org.apache.poi.hssf.usermodel.HSSFCell;
import org.apache.poi.hssf.usermodel.HSSFRow;
import org.apache.poi.hssf.usermodel.HSSFSheet;
import org.apache.poi.hssf.usermodel.HSSFWorkbook;

public class TestPOI3 {

  public static void main(
      String[] args) {

    HSSFWorkbook wb = new HSSFWorkbook();
    HSSFSheet sheet = wb.createSheet("ma feuille");

    HSSFRow row = sheet.createRow(0);
    HSSFCell cell = row.createCell((short)0);
    cell.setCellValue(10);

    row.createCell((short)1).setCellValue(20);

    FileOutputStream fileOut;
    try {
      fileOut = new FileOutputStream("monfichier.xls");
      wb.write(fileOut);
      fileOut.close();
    } catch (FileNotFoundException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```



Remarque : seules les lignes ayant un moins une cellule sont ajoutées à la feuille. Seules les cellules non vides sont ajoutées à une ligne.

La méthode setCellValue() possède plusieurs surcharges pour fournir une valeur à la cellule selon plusieurs formats : int, boolean, double et des objets de type Calendar, Date et chaîne de caractères.

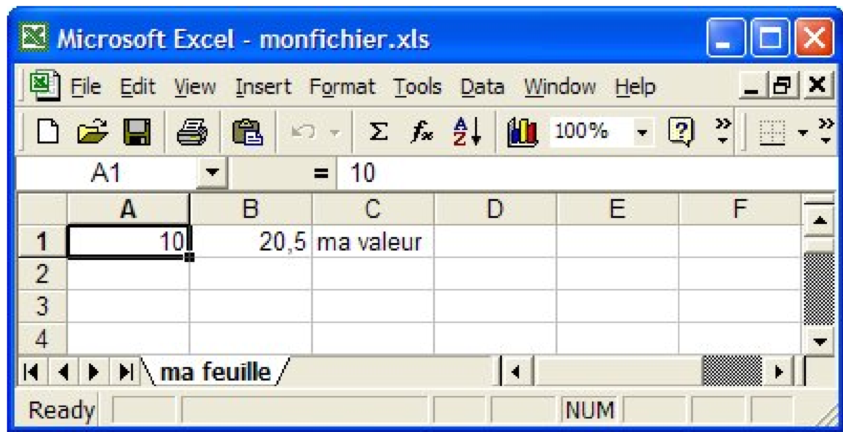
Remarque : pour les chaînes de caractères, la surcharge attendant en paramètre un objet de type String est deprecated au profit de la surcharge attend en paramètre un objet de type HSSFRichTextString.

La méthode setCellType() permet de préciser le type des données de la cellule. Elle attend en paramètre une des constantes définies dans la classe HSSFCell : CELL_TYPE_BLANK, CELL_TYPE_BOOLEAN, CELL_TYPE_ERROR, CELL_TYPE_FORMULA, CELL_TYPE_NUMERIC, ou CELL_TYPE_STRING

Exemple :

```
package com.jmdoudoux.test.poi;

import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

import org.apache.poi.hssf.usermodel.HSSFCell;
import org.apache.poi.hssf.usermodel.HSSFRichTextString;
import org.apache.poi.hssf.usermodel.HSSFRow;
import org.apache.poi.hssf.usermodel.HSSFSheet;
import org.apache.poi.hssf.usermodel.HSSFWorkbook;

public class TestPOI4 {

  public static void main(
      String[] args) {

    HSSFWorkbook wb = new HSSFWorkbook();
    HSSFSheet sheet = wb.createSheet("ma feuille");

    HSSFRow row = sheet.createRow(0);
    HSSFCell cell = row.createCell((short)0);
    cell.setCellValue(10);

    row.createCell((short)1).setCellValue(20.5);

    row.createCell((short)2, HSSFCell.CELL_TYPE_STRING)
      .setCellValue(new HSSFRichTextString("ma valeur"));;

    FileOutputStream fileOut;
    try {
      fileOut = new FileOutputStream("monfichier.xls");
      wb.write(fileOut);
      fileOut.close();
    } catch (FileNotFoundException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
```

```
    }
}
```



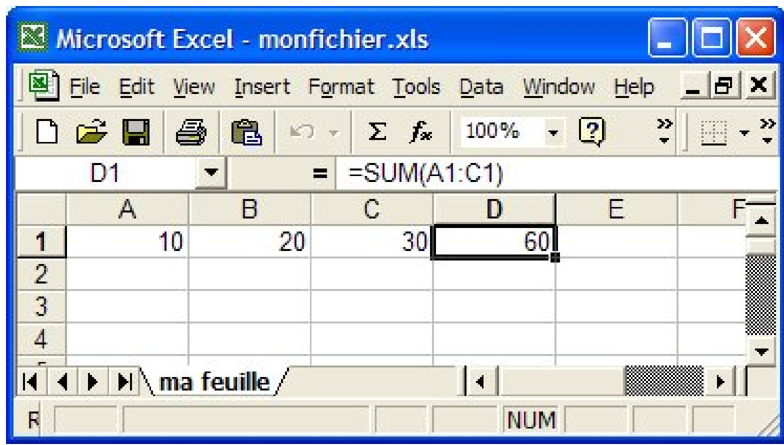
Une des grandes forces d'Excel est de permettre l'application de formules plus ou moins complexes sur les données.

Pour assigner une formule à une cellule, il faut lui assigner le type FORMULA. La méthode setCellFormula() permet de définir la formule qui sera associée à la cellule.

Exemple :

```
package com.jmdoudoux.test.poi;

import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

import org.apache.poi.hssf.usermodel.HSSFCell;
import org.apache.poi.hssf.usermodel.HSSFRow;
import org.apache.poi.hssf.usermodel.HSSFSheet;
import org.apache.poi.hssf.usermodel.HSSFWorkbook;

public class TestPOI13 {

  public static void main(
      String[] args) {

    HSSFWorkbook wb = new HSSFWorkbook();
    HSSFSheet sheet = wb.createSheet("ma feuille");

    HSSFRow row = sheet.createRow(0);
    HSSFCell cell = null;

    cell = row.createCell((short) 0);
    cell.setCellValue(10);

    cell = row.createCell((short) 1);
    cell.setCellValue(20);

    cell = row.createCell((short) 2);
    cell.setCellValue(30);

    cell = row.createCell((short) 3);
    cell.setCellType(HSSFCell.CELL_TYPE_FORMULA);
    cell.setCellFormula("SUM(A1:C1)");

    FileOutputStream fileOut;
    try {
      fileOut = new FileOutputStream("monfichier.xls");
      wb.write(fileOut);
      fileOut.close();
    } catch (FileNotFoundException e) {
      e.printStackTrace();
    } catch (IOException e) {
```

```
        e.printStackTrace();
      }
    }
}
```



82.1.1.1.4. Le formatage d'une cellule

Le formatage d'une cellule se fait à l'aide d'un objet de type HSSFCellStyle. La classe HSSFCellStyle permet de définir le format des données, aligner la valeur des données dans la cellule, définir des bordures autour de la cellule, ...

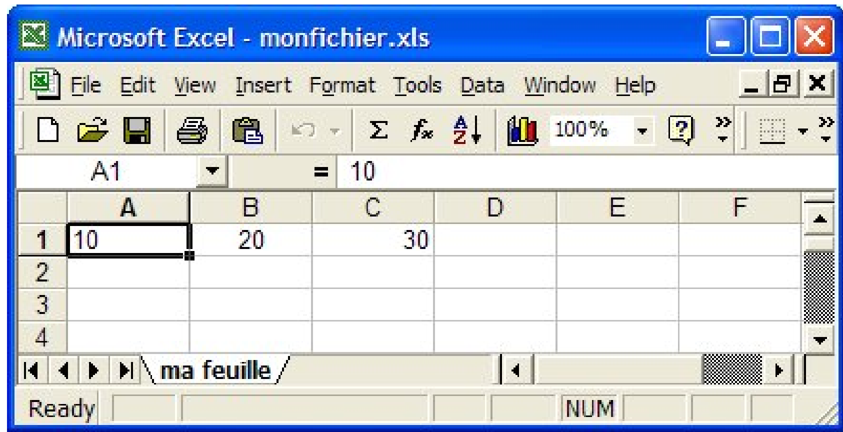
La méthode setCellStyle() de la classe HSSFCell permet d'associer un style à la cellule.

Pour définir un style qui permet d'aligner les données, il faut utiliser la méthode setAlignment() de la classe HSSFCellStyle. Celle-ci attend en paramètre une des constantes suivantes : HSSFCellStyle.ALIGN_CENTER, HSSFCellStyle.ALIGN_CENTER_SELECTION, HSSFCellStyle.ALIGN_FILL, HSSFCellStyle.ALIGN_GENERAL, HSSFCellStyle.ALIGN_JUSTIFY, HSSFCellStyle.ALIGN_LEFT, HSSFCellStyle.ALIGN_RIGHT

Exemple :

```
package com.jmdoudoux.test.poi;

import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

import org.apache.poi.hssf.usermodel.HSSFCell;
import org.apache.poi.hssf.usermodel.HSSFCellStyle;
import org.apache.poi.hssf.usermodel.HSSFRow;
import org.apache.poi.hssf.usermodel.HSSFSheet;
import org.apache.poi.hssf.usermodel.HSSFWorkbook;

public class TestPOI5 {

  public static void main(
      String[] args) {

    HSSFWorkbook wb = new HSSFWorkbook();
    HSSFSheet sheet = wb.createSheet("ma feuille");

    HSSFRow row = sheet.createRow(0);
    HSSFCell cell = null;
    HSSFCellStyle cellStyle = null;

    cell = row.createCell((short) 0);
    cell.setCellValue(10);
    cellStyle = wb.createCellStyle();
    cellStyle.setAlignment(HSSFCellStyle.ALIGN_LEFT);
    cell.setCellStyle(cellStyle);

    cell = row.createCell((short) 1);
```

```
    cell.setCellValue(20);
    cellStyle = wb.createCellStyle();
    cellStyle.setAlignment(HSSFCellStyle.ALIGN_CENTER);
    cell.setCellStyle(cellStyle);

    cell = row.createCell((short) 2);
    cell.setCellValue(30);
    cellStyle = wb.createCellStyle();
    cellStyle.setAlignment(HSSFCellStyle.ALIGN_RIGHT);
    cell.setCellStyle(cellStyle);

    FileOutputStream fileOut;
    try {
      fileOut = new FileOutputStream("monfichier.xls");
      wb.write(fileOut);
      fileOut.close();
    } catch (FileNotFoundException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```



Pour préciser le format des données de la cellule, il faut utiliser la méthode setDataFormat() de la classe HSSFCellStyle. Elle attend en paramètre un entier qui précise le type selon les valeurs gérées par la classe HSSFDataFormat.

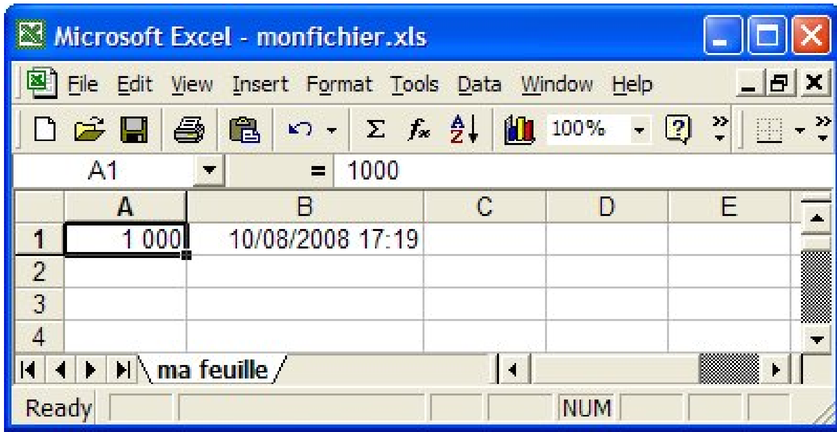
La classe HSSFDataFormat propose plusieurs méthodes statiques pour obtenir un des formatages prédéfinis.

Pour utiliser un format personnalisé, il faut invoquer la méthode createDataFormat() de la classe HSSFWorkbook de l'instance qui encapsule le document pour obtenir une instance de la classe HSSFDataFormat. L'invocation de la méthode getFormat() de cette instance permet de définir son format personnalisé.

Exemple :

```
package com.jmdoudoux.test.poi;

import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.Date;

import org.apache.poi.hssf.usermodel.HSSFCell;
import org.apache.poi.hssf.usermodel.HSSFCellStyle;
import org.apache.poi.hssf.usermodel.HSSFDataFormat;
import org.apache.poi.hssf.usermodel.HSSFRow;
import org.apache.poi.hssf.usermodel.HSSFSheet;
import org.apache.poi.hssf.usermodel.HSSFWorkbook;

public class TestPOI6 {

  public static void main(
      String[] args) {
```

```
    HSSFWorkbook wb = new HSSFWorkbook();
    HSSFSheet sheet = wb.createSheet("ma feuille");

    HSSFRow row = sheet.createRow(0);
    HSSFCell cell = null;
    HSSFCellStyle cellStyle = null;

    cell = row.createCell((short) 0);
    cell.setCellValue(1000);
    cellStyle = wb.createCellStyle();
    cellStyle.setDataFormat(HSSFDataFormat.getBuiltinFormat("#,##0"));
    cell.setCellStyle(cellStyle);

    cell = row.createCell((short) 1);
    cell.setCellValue(new Date());
    cellStyle = wb.createCellStyle();
    HSSFDataFormat hssfDataFormat = wb.createDataFormat();
    cellStyle.setDataFormat(hssfDataFormat.getFormat("dd/mm/yyyy h:mm"));
    cell.setCellStyle(cellStyle);

    FileOutputStream fileOut;
    try {
      fileOut = new FileOutputStream("monfichier.xls");
      wb.write(fileOut);
      fileOut.close();
    } catch (FileNotFoundException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```



Pour modifier le fond et l'apparence d'une cellule, il faut utiliser les méthodes setFillBackgroundColor(), setFillForegroundColor() et setFillPattern() de la classe HSSFCellStyle.

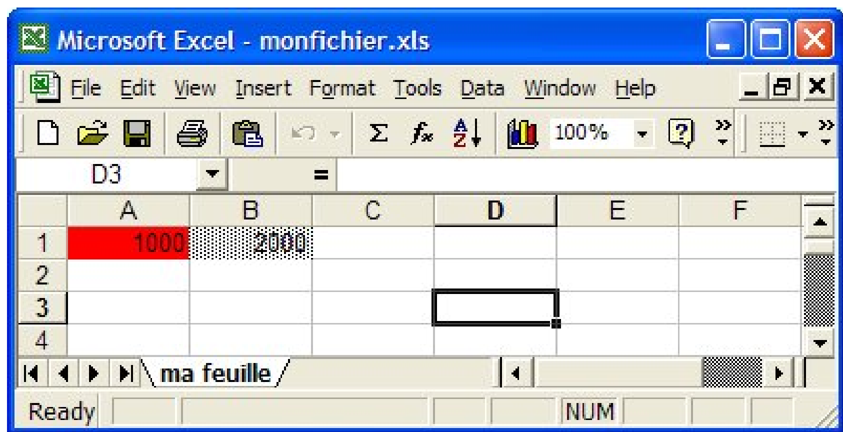
Les méthodes setFillBackgroundColor() et setFillForegroundColor() attendent en paramètre un entier de type short : la classe HSSFColor possède de nombreuses classes filles pour faciliter l'utilisation d'une couleur.

La méthode setFillPattern() attend en paramètre un entier de type short : la classe HSSFCellStyle propose de nombreuses constantes pour faciliter l'utilisation d'un motif de remplissage.

Exemple :

```
package com.jmdoudoux.test.poi;

import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

import org.apache.poi.hssf.usermodel.HSSFCell;
import org.apache.poi.hssf.usermodel.HSSFCellStyle;
```

```
import org.apache.poi.hssf.usermodel.HSSFRow;
import org.apache.poi.hssf.usermodel.HSSFSheet;
import org.apache.poi.hssf.usermodel.HSSFWorkbook;
import org.apache.poi.hssf.util.HSSFColor;

public class TestPOI7 {

  public static void main(
      String[] args) {

    HSSFWorkbook wb = new HSSFWorkbook();
    HSSFSheet sheet = wb.createSheet("ma feuille");

    HSSFRow row = sheet.createRow(0);
    HSSFCell cell = null;
    HSSFCellStyle cellStyle = null;

    cell = row.createCell((short) 0);
    cell.setCellValue(1000);
    cellStyle = wb.createCellStyle();
    cellStyle.setFillForegroundColor(HSSFColor.RED.index);
    cellStyle.setFillPattern(HSSFCellStyle.SOLID_FOREGROUND);
    cell.setCellStyle(cellStyle);

    cell = row.createCell((short) 1);
    cell.setCellValue(2000);
    cellStyle = wb.createCellStyle();
    cellStyle.setFillForegroundColor(HSSFColor.YELLOW.index);
    cellStyle.setFillPattern(HSSFCellStyle.ALT_BARS);
    cellStyle.setFillForegroundColor(HSSFColor.WHITE.index);
    cell.setCellStyle(cellStyle);

    FileOutputStream fileOut;
    try {
      fileOut = new FileOutputStream("monfichier.xls");
      wb.write(fileOut);
      fileOut.close();
    } catch (FileNotFoundException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

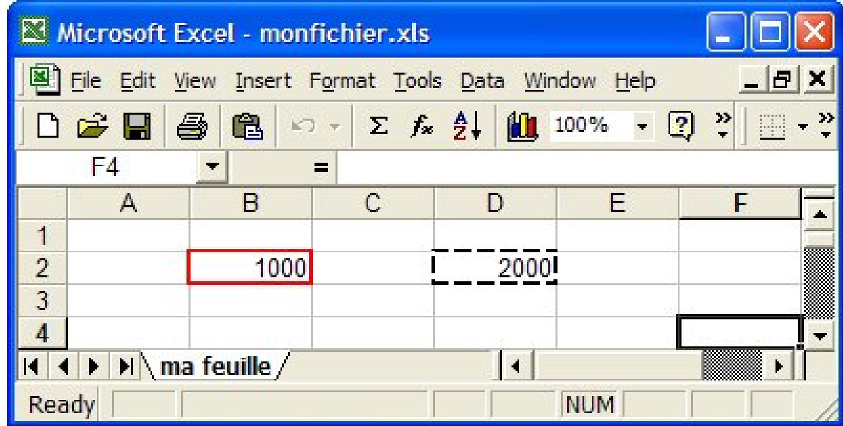


Pour mettre des bordures sur les côtés des cellules, la clase HSSFCellStyle propose plusieurs méthodes :

- setBorderXxx : permet de préciser la forme de la bordure en utilisant les constantes définies dans la classe HSSFCellStyle
- setXxxBorderColor : permet de préciser la couleur de la bordure

Exemple :

```
package com.jmdoudoux.test.poi;

import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

import org.apache.poi.hssf.usermodel.HSSFCell;
import org.apache.poi.hssf.usermodel.HSSFCellStyle;
import org.apache.poi.hssf.usermodel.HSSFRow;
import org.apache.poi.hssf.usermodel.HSSFSheet;
import org.apache.poi.hssf.usermodel.HSSFWorkbook;
import org.apache.poi.hssf.util.HSSFColor;

public class TestPOI8 {

  public static void main(
      String[] args) {

    HSSFWorkbook wb = new HSSFWorkbook();
    HSSFSheet sheet = wb.createSheet("ma feuille");

    HSSFRow row = sheet.createRow(1);
    HSSFCell cell = null;
    HSSFCellStyle cellStyle = null;

    cell = row.createCell((short) 1);
    cell.setCellValue(1000);
    cellStyle = wb.createCellStyle();
    cellStyle.setBorderBottom(HSSFCellStyle.BORDER_MEDIUM);
    cellStyle.setBottomBorderColor(HSSFColor.RED.index);
    cellStyle.setBorderLeft(HSSFCellStyle.BORDER_MEDIUM);
    cellStyle.setLeftBorderColor(HSSFColor.RED.index);
    cellStyle.setBorderRight(HSSFCellStyle.BORDER_MEDIUM);
    cellStyle.setRightBorderColor(HSSFColor.RED.index);
    cellStyle.setBorderTop(HSSFCellStyle.BORDER_MEDIUM);
    cellStyle.setTopBorderColor(HSSFColor.RED.index);
    cell.setCellStyle(cellStyle);

    cell = row.createCell((short) 3);
    cell.setCellValue(2000);
    cellStyle = wb.createCellStyle();
    cellStyle.setBorderBottom(HSSFCellStyle.BORDER_MEDIUM_DASHED);
    cellStyle.setBottomBorderColor(HSSFColor.BLACK.index);
    cellStyle.setBorderLeft(HSSFCellStyle.BORDER_MEDIUM_DASHED);
    cellStyle.setLeftBorderColor(HSSFColor.BLACK.index);
    cellStyle.setBorderRight(HSSFCellStyle.BORDER_MEDIUM_DASHED);
    cellStyle.setRightBorderColor(HSSFColor.BLACK.index);
    cellStyle.setBorderTop(HSSFCellStyle.BORDER_MEDIUM_DASHED);
    cellStyle.setTopBorderColor(HSSFColor.BLACK.index);
    cell.setCellStyle(cellStyle);

    FileOutputStream fileOut;
    try {
      fileOut = new FileOutputStream("monfichier.xls");
      wb.write(fileOut);
      fileOut.close();
    } catch (FileNotFoundException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```



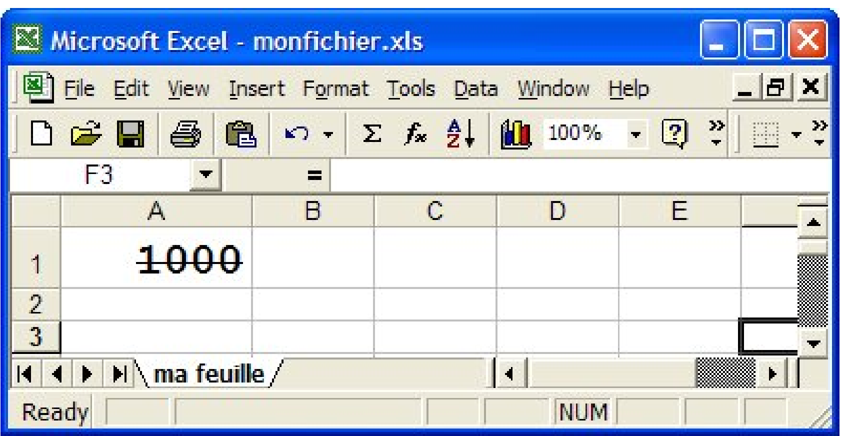
Pour utiliser une police de caractères, il faut utiliser la méthode setFont() de la classe HSSFCellStyle qui attend en paramètre un objet de type HSSFFont qui encapsule une police de caractères.

Pour obtenir une instance de la classe HSSFFont, il faut invoquer la méthode createFont() de la classe HSSFWorkbook. La classe HSSFFont possède plusieurs méthodes pour définir les caractéristiques de la police de caractères : famille, taille, gras, souligné, italique, barré, ...

Exemple :

```
package com.jmdoudoux.test.poi;

import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

import org.apache.poi.hssf.usermodel.HSSFCell;
import org.apache.poi.hssf.usermodel.HSSFCellStyle;
import org.apache.poi.hssf.usermodel.HSSFFont;
import org.apache.poi.hssf.usermodel.HSSFRow;
import org.apache.poi.hssf.usermodel.HSSFSheet;
import org.apache.poi.hssf.usermodel.HSSFWorkbook;

public class TestPOI9 {

  public static void main(
      String[] args) {

    HSSFWorkbook wb = new HSSFWorkbook();
    HSSFSheet sheet = wb.createSheet("ma feuille");

    HSSFRow row = sheet.createRow(0);
    HSSFCell cell = null;
    HSSFCellStyle cellStyle = null;

    HSSFFont fonte = wb.createFont();
    fonte.setFontHeightInPoints((short) 18);
    fonte.setFontName("Courier New");
    fonte.setBoldweight(HSSFFont.BOLDWEIGHT_BOLD);
    fonte.setStrikeout(true);

    cell = row.createCell((short) 0);
    cell.setCellValue(1000);
    cellStyle = wb.createCellStyle();
    cellStyle.setFont(fonte);
    cell.setCellStyle(cellStyle);

    FileOutputStream fileOut;
    try {
      fileOut = new FileOutputStream("monfichier.xls");
      wb.write(fileOut);
      fileOut.close();
    } catch (FileNotFoundException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
```

```
        }
    }
}
```



82.1.1.1.5. La fusion de cellules

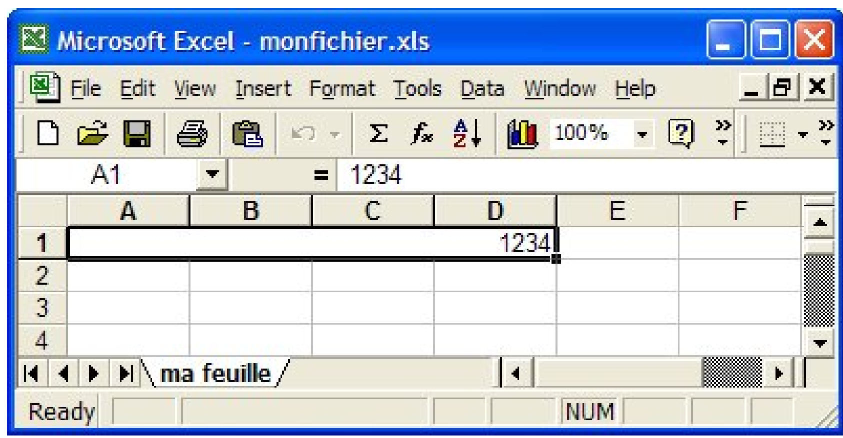
Pour fusionner un ensemble de cellules, il faut invoquer la méthode addMergedRegion() de la classe HSSFSheet. Elle attend en paramètre un objet de type org.apache.poi.hssf.util.Region qui permet de définir l'ensemble des colonnes à fusionner.

Un des constructeurs de la classe Region attend en paramètre quatre entiers qui correspondent respectivement au numéro de la première ligne, à la colonne de la première ligne, au numéro de la dernière ligne et à la colonne de la dernière ligne de l'ensemble des cellules.

Exemple :

```
package com.jmdoudoux.test.poi;

import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

import org.apache.poi.hssf.usermodel.HSSFCell;
import org.apache.poi.hssf.usermodel.HSSFRow;
import org.apache.poi.hssf.usermodel.HSSFSheet;
import org.apache.poi.hssf.usermodel.HSSFWorkbook;
import org.apache.poi.hssf.util.Region;

public class TestPOI10 {

  public static void main(
      String[] args) {

    HSSFWorkbook wb = new HSSFWorkbook();
    HSSFSheet sheet = wb.createSheet("ma feuille");

    HSSFRow row = sheet.createRow(0);
    HSSFCell cell = null;

    cell = row.createCell((short) 0);
    cell.setCellValue(1234);

    sheet.addMergedRegion(new Region(0,(short)0,0,(short)3));

    FileOutputStream fileOut;
    try {
      fileOut = new FileOutputStream("monfichier.xls");
      wb.write(fileOut);
      fileOut.close();
    } catch (FileNotFoundException e) {
```

```
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```



82.1.1.1.6. La lecture et la modification d'un document

Pour lire un document, il faut utiliser la classe POIFSFileSystem. Un des constructeurs de cette classe attend en paramètre un objet de type InputStream qui encapsule un flux vers le document à lire.

Il suffit de fournir l'instance de la classe POIFSFileSystem en paramètre du constructeur de la classe HSSFWorkbook pour lire le document et une arborescence d'objets qui encapsule son contenu.

Il est ensuite possible d'utiliser ces objets pour modifier le contenu du document et de l'enregistrer une fois les modifications terminées.

Les méthodes getNumericCellValue() et getRichStringCellValue() de la classe HSSFCell permettent d'obtenir la valeur de la cellule selon son type.

La méthode setCellType() de la classe HSSFCell permet de préciser le type du contenu de la cellule (CELL_TYPE_BLANK, CELL_TYPE_BOOLEAN, CELL_TYPE_ERROR, CELL_TYPE_FORMULA, CELL_TYPE_NUMERIC, CELL_TYPE_STRING)

Plusieurs surcharges de la méthode setValue()de la classe HSSFCell permettent de fournir la valeur de la cellule.

Exemple :

```
package com.jmdoudoux.test.poi;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

import org.apache.poi.hssf.usermodel.HSSFCell;
import org.apache.poi.hssf.usermodel.HSSFRichTextString;
import org.apache.poi.hssf.usermodel.HSSFRow;
import org.apache.poi.hssf.usermodel.HSSFSheet;
import org.apache.poi.hssf.usermodel.HSSFWorkbook;
import org.apache.poi.poifs.filesystem.POIFSFileSystem;

public class TestPOI11 {

  public static void main(
      String[] args) {
```

```
    try {
      FileOutputStream fileOut;

      POIFSFileSystem fs = new POIFSFileSystem(new FileInputStream("monfichier.xls"));
      HSSFWorkbook wb = new HSSFWorkbook(fs);
      HSSFSheet sheet = wb.getSheetAt(0);
      HSSFRow row = sheet.getRow(0);

      HSSFCell cell = row.getCell((short) 0);
      if (cell != null)
          row.removeCell(cell);
      cell = row.createCell((short) 0);
      cell.setCellType(HSSFCell.CELL_TYPE_STRING);
      cell.setCellValue(new HSSFRichTextString("données modifiées"));

      fileOut = new FileOutputStream("monfichier.xls");
      wb.write(fileOut);
      fileOut.close();
    } catch (FileNotFoundException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Pour obtenir une donnée, il est préférable de s'assurer du type de données associé à la cellule en utilisant la méthode getCellType().

Une exception est levée si le type de données demandé ne correspond pas à la méthode invoquée : exemple l'appel de la méthode getCellValue() sur une cellule de type STRING.

Remarque : Excel stocke les dates sous une forme numérique. Pour les identifier, il faut regarder le format des données.

82.1.1.1.7. Le parcours des cellules d'une feuille

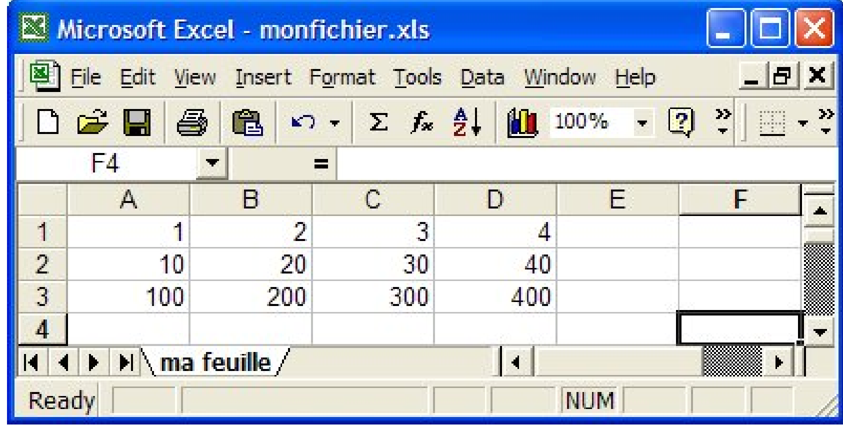
La classe HSSFSheet propose la méthode rowIterator() qui renvoie un objet de type Iterator qui permet de parcourir les lignes de la feuille.

Attention : seules les lignes non vides sont contenues dans l'iterator. Ainsi il n'y pas de corrélation entre le numéro de la ligne de l'iterator et le numéro de la ligne dans la feuille. Pour connaître le numéro de la ligne dans la feuille, il faut utiliser la méthode getRowNum() de la classe HSSFRow.

La classe HSSFRow propose la méthode cellIterator() qui renvoie un objet de type Iterator qui permet de parcourir les cellules de la ligne.

Attention : seules les cellules non vides sont contenues dans l'iterator. Ainsi il n'y pas de corrélation entre le numéro de la cellule de l'iterator et le numéro de la cellule dans la ligne. Pour connaître le numéro de la colonne dans la ligne, il faut utiliser la méthode getCellNum() de la classe HSSFCell.

L'exemple ci-dessous va utiliser le fichier suivant



L'application va parcourir les cellules et afficher le total de chaque ligne et le total de toutes les cellules.

Exemple :

```
package com.jmdoudoux.test.poi;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.Iterator;

import org.apache.poi.hssf.usermodel.HSSFCell;
import org.apache.poi.hssf.usermodel.HSSFRow;
import org.apache.poi.hssf.usermodel.HSSFSheet;
import org.apache.poi.hssf.usermodel.HSSFWorkbook;
import org.apache.poi.poifs.filesystem.POIFSFileSystem;

public class TestPOI12 {

  public static void main(
      String[] args) {

    try {
      POIFSFileSystem fs = new POIFSFileSystem(new FileInputStream("monfichier.xls"));
      HSSFWorkbook wb = new HSSFWorkbook(fs);
      HSSFSheet sheet = wb.getSheetAt(0);
      HSSFRow row = null;
      HSSFCell cell = null;
      double totalLigne = 0.0;
      double totalGeneral = 0.0;
      int numLigne = 1;

      for (Iterator rowIt = sheet.rowIterator(); rowIt.hasNext();) {
        totalLigne = 0;
        row = (HSSFRow) rowIt.next();
        for (Iterator cellIt = row.cellIterator(); cellIt.hasNext();) {
          cell = (HSSFCell) cellIt.next();
          totalLigne += cell.getNumericCellValue();
        }
        System.out.println("total ligne "+numLigne+" = "+totalLigne);
        totalGeneral += totalLigne;
        numLigne++;
      }
      System.out.println("total general "+totalGeneral);

    } catch (FileNotFoundException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
total ligne 1 = 10.0
total ligne 2 = 100.0
total ligne 3 = 1000.0
total general 1110.0
```

Il est aussi possible d'utiliser les generics en utilisant Java 5.

Exemple :

```
package com.jmdoudoux.test.poi;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.Iterator;

import org.apache.poi.hssf.usermodel.HSSFCell;
import org.apache.poi.hssf.usermodel.HSSFRow;
import org.apache.poi.hssf.usermodel.HSSFSheet;
import org.apache.poi.hssf.usermodel.HSSFWorkbook;
import org.apache.poi.poifs.filesystem.POIFSFileSystem;

public class TestPOI12 {

  public static void main(
      String[] args) {

    try {
      POIFSFileSystem fs = new POIFSFileSystem(new FileInputStream("monfichier.xls"));
      HSSFWorkbook wb = new HSSFWorkbook(fs);
      HSSFSheet sheet = wb.getSheetAt(0);
      HSSFRow row = null;
      HSSFCell cell = null;
      double totalLigne = 0.0;
      double totalGeneral = 0.0;
      int numLigne = 1;

      for (Iterator<HSSFRow> rowIt = (Iterator<HSSFRow>) sheet.rowIterator();
        rowIt.hasNext();) {
        totalLigne = 0;
        row = rowIt.next();
        for (Iterator<HSSFCell> cellIt = (Iterator<HSSFCell>) row.cellIterator();
          cellIt.hasNext();) {
          cell = cellIt.next();
          totalLigne += cell.getNumericCellValue();
        }
        System.out.println("total ligne "+numLigne+" = "+totalLigne);
        totalGeneral += totalLigne;
        numLigne++;
      }
      System.out.println("total general "+totalGeneral);

    } catch (FileNotFoundException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

82.1.1.1.8. La génération d'un document Excel dans une servlet

Il peut être utile de faire générer un document Excel par une servlet pour que celle-ci retourne le document dans sa réponse.

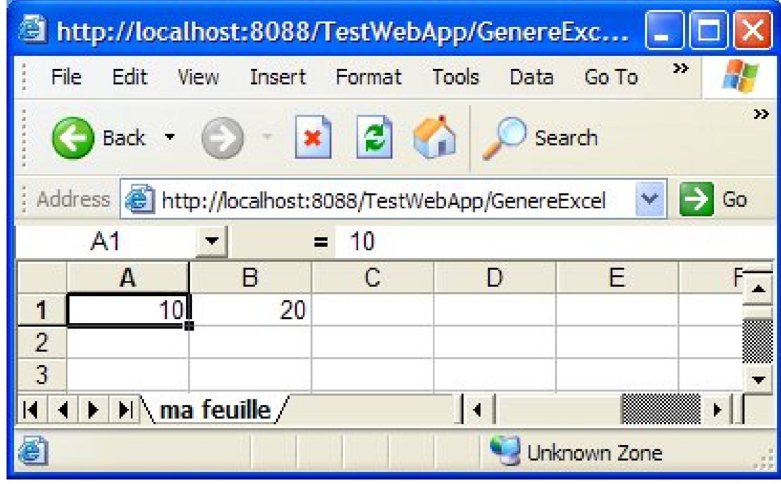
Il est nécessaire de correctement positionner le type mime sur "application/vnd.ms-excel" qui désigne l'application Excel.

Il est aussi utile de définir la propriété "Content-disposition" pour faciliter l'enregistrement du document par l'utilisateur.

Enfin, il faut simplement fournir en paramètre de la méthode write() de la classe HSSFWorkbook le flux de sortie de la réponse HTTP de la servlet.

Exemple :

```
package com.jmdoudoux.test.poi;

import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.OutputStream;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.poi.hssf.usermodel.HSSFCell;
import org.apache.poi.hssf.usermodel.HSSFRow;
import org.apache.poi.hssf.usermodel.HSSFSheet;
import org.apache.poi.hssf.usermodel.HSSFWorkbook;

public class GenereExcel extends javax.servlet.http.HttpServlet
  implements javax.servlet.Servlet {

  public GenereExcel() {
    super();
  }

  protected void doGet(
      HttpServletRequest request,
      HttpServletResponse response) throws ServletException, IOException {
    try {
      OutputStream out = response.getOutputStream();

      response.setContentType("application/vnd.ms-excel");

      response.setHeader("Content-disposition", "inline; filename=monfichier.xls");
      HSSFWorkbook wb = new HSSFWorkbook();

      HSSFSheet sheet = wb.createSheet("ma feuille");

      HSSFRow row = sheet.createRow(0);
      HSSFCell cell = row.createCell((short) 0);
      cell.setCellValue(10);

      row.createCell((short) 1).setCellValue(20);

      wb.write(out);
      out.flush();
      out.close();
    } catch (FileNotFoundException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```



82.1.1.2. L'API de type eventusermodel

L'utilisation de cette API est particulièrement adaptée à la lecture de gros fichiers Excel car elle ne charge pas le document en mémoire mais émet des événements lors de la lecture du document. Cette API permet uniquement la lecture de document.

Sa mise en oeuvre n'est cependant pas très implicite et nécessite quelques notions sur la structure de bas niveau du document Excel.

Consultez la documentation de POI-HSSF pour le détail de sa mise en oeuvre.

82.2. iText

iText est une API open source qui permet la génération de documents PDF, RTF et HTML. Elle est diffusée sous deux licences : MPL et LGPL.

Le site officiel de cette API est à l'url : http://www.lowagie.com/iText/

iText contient de très nombreuses classes permettant de réaliser de nombreuses actions basiques et avancées notamment pour la génération de documents de type PDF.

iText est une API qui permet d'intégrer dans une application la génération dynamique de documents : ceci est particulièrement utile lorsque le contenu du document dépend d'informations fournies ou obtenues à partir d'informations de l'utilisateur ou contient des données calculées.

La possibilité d'iText d'exporter un document créé avec l'API dans différents formats peut être pratique. Le document exporté peut en outre être envoyé vers différents flux (fichier, réponse http d'une servlet, console, ...).

iText permet aussi la mise en oeuvre de fonctionnalités avancées sur un document PDF :

- définition de marque pages, filigranes, ...
- signature numérique
- remplissage de formulaires
- diviser un document ou assembler plusieurs documents
- ...

iText requiert un JDK 1.4 minimum et l'API BouncyCastle pour certaines fonctionnalités.

82.2.1. Un exemple très simple

L'exemple proposé va créer un document PDF qui contient "Hello World".

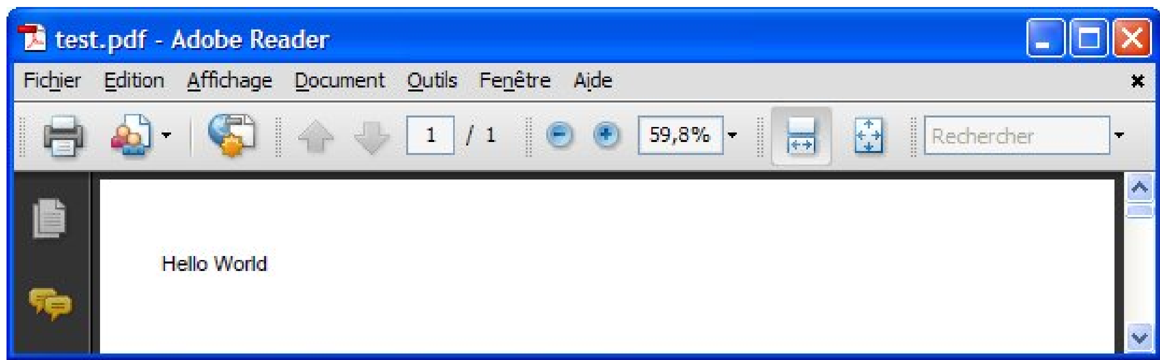
La création d'un document PDF avec iText se fait en cinq étapes :

- Instanciation d'un objet de type Document
- Instanciation d'un objet de type PdfWriter pour exporter le document
- Appel de la méthode open() du document
- Création et ajout des éléments qui composent le document
- Appel de la méthode close() pour exporter le document

Exemple :

```
package com.jmdoudoux.test.itext;

import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.PageSize;
import com.lowagie.text.Paragraph;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText1 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter.getInstance(document,
          new FileOutputStream("c:/test.pdf"));
      document.open();
      document.add(new Paragraph("Hello World"));
    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    document.close();
  }
}
```

Pour compiler et exécuter cet exemple, il faut ajouter la bibliothèque iText-2.1.3.jar au classpath.



82.2.2. L'API de iText

L'API de iText contient des objets qui proposent des fonctionnalités pour la création de documents notamment au format PDF. Ils encapsulent par exemple le document et chacun des éléments qui peuvent le composer tel que le objets de type

Font, Paragraph, Chapter, Anchor, Image, List, Table, ...

L'API de iText de composée de nombreuses classes : seules les principales sont présentées dans cette section. Le site officiel et la Javadoc de l'API fournissent des informations détaillées sur l'ensemble des fonctionnalités de chacune des classes.

82.2.3. La création d'un document

La création d'un document avec iText se fait en plusieurs étapes :

- Instanciation d'un objet de type Document
- Instanciation d'un objet de type Writer pour exporter le document
- Appel de la méthode open() du document
- Création et ajout des éléments qui composent le document
- Appel de la méthode close() pour exporter le document

82.2.3.1. La classe Document

La classe Document est un conteneur pour le contenu d'un document.

L'objet Document possède plusieurs constructeurs :

Constructeur	Rôle
Document()	constructeur par défaut
Document(Rectangle pageSize)	constructeur qui précise la taille des pages du document
Document(Rectangle pageSize, int marginLeft, int marginRight, int marginTop, int marginBottom)	constructeur qui précise la taille des pages du document et leurs marges

La taille des pages peut être définies en utilisant deux des surcharges du constructeur ou en utilisant la méthode setPageSize().

Un objet de type Rectangle permet de définir la taille des pages du document.

L'unité de mesure dans un document est le point. Il y a 72 points dans un pouce et un pouce vaut 2,54 cm. Ainsi par exemple, la taille d'une page A4 vaut :

largeur : (21 / 2,54) * 72 = 595 points

hauteur : (29,7 / 2,54) * 72 = 842 points

La classe PageSize définit de nombreuses constantes de type Rectangle pour les tailles de pages standards (A0 à A10, LETTER, LEGAL, ...).

Par défaut, la taille utilisée est PageSize.A4.

La plupart des tailles prédéfinies sont au format portrait. Pour utiliser une taille au format paysage, il faut utiliser la méthode rotate() de la classe Rectangle.

Exemple :

```
...
      Document document = new Document(PageSize.A4.rotate());
...
```

La marge par défaut est de 36 points. La marge par défaut peut être précisée dans la surcharge du constructeur dédiée ou en utilisant la méthode setMargins(). Durant la création du contenu d'un document, il est possible d'utiliser la méthode setMargins() pour modifier les marges : cette modification ne sera effective qu'à partir de la page suivante.

La mise en oeuvre d'un document impose quelques contraintes :

- les méta données doivent impérativement être associées au document avant l'appel de la méthode open()
- il n'est possible d'ajouter le contenu du document qu'une fois que la méthode open() est été invoquée
- la modification de l'en-tête et du pied page n'est effective qu'à partir de la page suivante

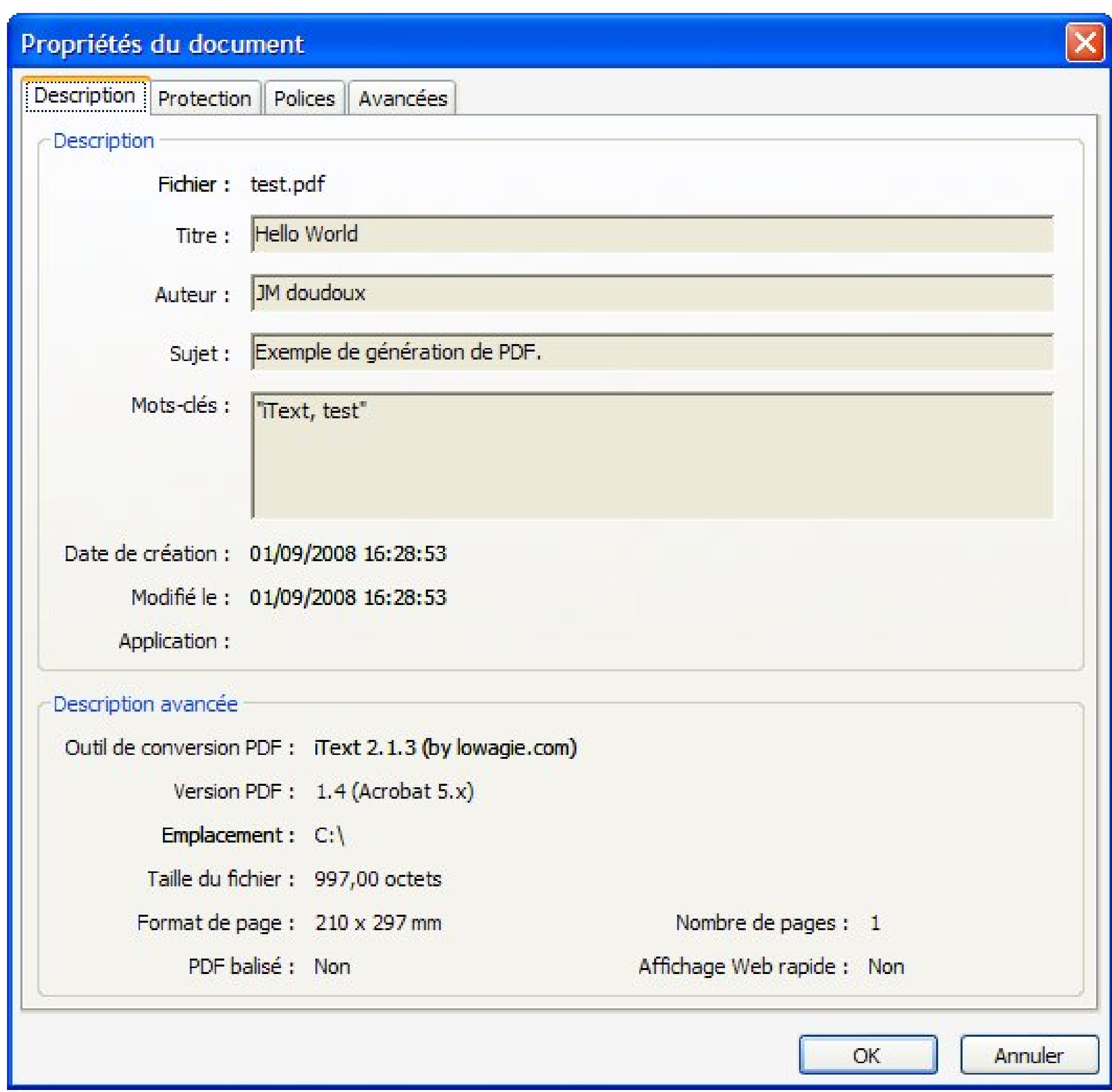
La classe Document possède plusieurs méthodes pour associer des méta-données au document :

Méthode	Rôle
boolean addTitle(String title)	ajout du titre du document fourni en paramètre
boolean addSubject(String subject)	ajout du sujet du document fourni en paramètre
boolean addKeywords(String keywords)	ajout du mot clé fourni en paramètre
boolean addAuthor(String author)	ajout de l'auteur fourni en paramètre
boolean addCreator(String creator)	ajout du créateur fourni en paramètre
boolean addProducer()	ajout iText comme outil de production du document
boolean addCreationDate()	ajout de la date actuelle comme date de création
boolean addHeader(String name, String content)	ajout d'une méta donnée personnalisée

Remarque : l'utilisation de la méthode addHeader() n'a pas d'effet si le document est exporté en PDF.

Exemple :

```
package com.jmdoudoux.test.itext;

import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.PageSize;
import com.lowagie.text.Paragraph;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText3 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter.getInstance(document,
          new FileOutputStream("c:/test.pdf"));
      document.addTitle("Hello World");
      document.addAuthor("JM doudoux");
      document.addSubject("Exemple de génération de PDF.");
      document.addKeywords("iText, test");
      document.open();
      document.add(new Paragraph("Hello World"));
    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    document.close();
  }
}
```



Important : l'ajout de méta-données doit obligatoirement se faire avant l'appel à la méthode open().

Avant de pouvoir ajouter du contenu au document, il faut obligatoirement invoquer la méthode open() de l'instance de la classe Document. Dans le cas contraire, une exception de type DocumentException est levée.

Exemple :

```
package com.jmdoudoux.test.itext;

import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.PageSize;
import com.lowagie.text.Paragraph;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText2 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter.getInstance(document,
          new FileOutputStream("c:/test.pdf"));
      document.add(new Paragraph("Hello World"));
    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
```

```
    }

    document.close();
  }
}
```

Résultat :

```
com.lowagie.text.DocumentException: The document is not open yet; you can only add Meta
information.
               at com.lowagie.text.Document.add(Unknown Source)
               at com.jmdoudoux.test.itext.TestIText2.main(TestIText2.java:20)
Exception in thread "main" java.lang.RuntimeException: The document is not open.
               at com.lowagie.text.pdf.PdfWriter.getDirectContent(Unknown Source)
               at com.lowagie.text.pdf.PdfDocument.newPage(Unknown Source)
               at com.lowagie.text.pdf.PdfDocument.close(Unknown Source)
               at com.lowagie.text.Document.close(Unknown Source)
               at com.jmdoudoux.test.itext.TestIText2.main(TestIText2.java:27)
```

Avant de pouvoir ajouter du contenu au document, il faut impérativement invoquer la méthode open().

La méthode add() permet d'ajouter un élément au contenu du document.

Il est important d'invoquer la méthode close() du document une fois celui-ci complet : la méthode close() va demander la fermeture du ou des flux d'exportation du document.

Attention : la classe Document encapsule le contenu du document mais ne contient aucune information sur le rendu du document. Le rendu est assuré par différents writers (par exemple un document peut avoir plusieurs pages en PDF mais une seule en HTML) : ainsi il ne faut pas utiliser la méthode getPageNumber() de la classe Document.

82.2.3.2. Les objets de type DocWriter

Pour exporter le document, il faut lui associer un ou plusieurs DocWriters. Chaque DocWriter permet l'exportation du document dans un format particulier.

Trois classes héritent de la classe DocWriter :

- PdfWriter : permet l'exportation d'un document au format PDF
- HtmlWriter : permet l'exportation d'un document au format HTML
- RtfWriter2 : permet l'exportation d'un document au format RTF

Pour obtenir une instance d'un objet héritant du type DocWriter, il faut utiliser sa méthode statique getInstance() qui attend en paramètre l'instance du document et le flux vers lequel le document sera exporté. Ce flux peut être de différents types selon les besoins : FileOutputStream pour un fichier, ServletOutputStream pour une réponse d'une servlet, ByteArrayOutputStream pour stocker le document en mémoire, ...

Il est possible d'affecter plusieurs DocWriter à un document utilisant des flux différents.

Il est nécessaire de conserver l'instance retournée par la méthode getInstance() pour mettre en oeuvre quelques fonctionnalités avancées de l'exportation.

82.2.3.2.1. La classe PdfWriter

La classe PdfWriter permet l'exportation d'un document au format PDF.

Elle propose de nombreuses méthodes permettant de définir des caractéristiques spécifiques au format PDF.

La méthode setViewerPreferences() permet de préciser le mode d'affichage du document par défaut. Elle attend en paramètre un entier pour lequel plusieurs constantes sont définies.

Plusieurs constantes peuvent être combinées pour préciser le mode d'affichage des éléments du panneau de navigation.

Constante	Rôle
PdfWriter.PageModeFullScreen	affichage en plein écran
PdfWriter.PageModeUseAttachments	afficher les pièces jointes
PdfWriter.PageModeUseNone	n'afficher aucun élément du panneau de navigation
PdfWriter.PageModeUseOC	afficher les calques
PdfWriter.PageModeUseOutlines	afficher l'arborescence
PdfWriter.PageModeUseThumbs	affichage des vignettes

Plusieurs autres constantes peuvent être combinées pour préciser le mode d'affichage des pages.

Constante	Rôle
PdfWriter.PageLayoutSinglePage	affichage d'une seule page à la fois
PdfWriter.PageLayoutOneColumn	affichage des pages dans une colonne
PdfWriter.PageLayoutTwoColumnLeft	affichage des pages dans deux colonnes de gauche à droite
PdfWriter.PageLayoutTwoColumnRight	affichage des pages dans deux colonnes (de droite à gauche)

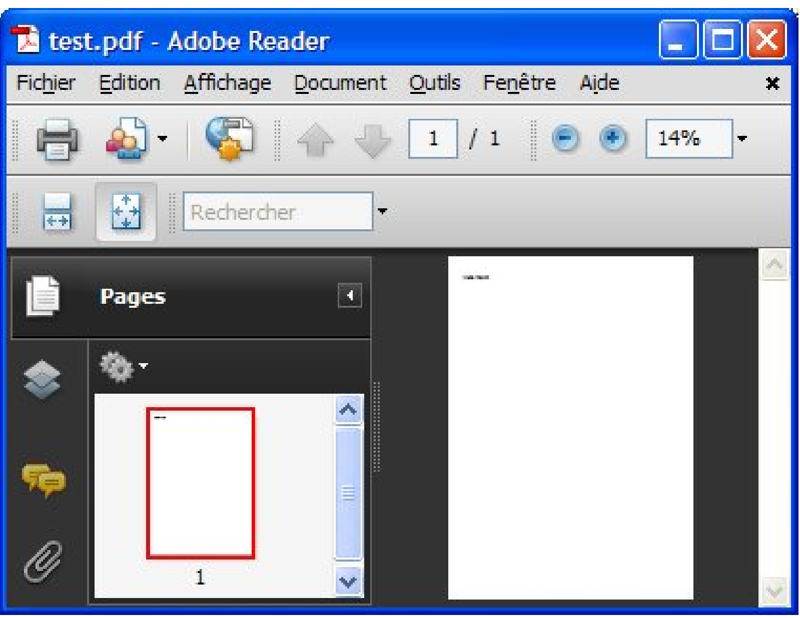
Plusieurs autres constantes peuvent être combinées pour afficher ou non quelques éléments de l'interface graphique d'Adobe Reader.

Constante	Rôle
PdfWriter.HideToolBar	permet de spécifier si la barre d'outils est affichée
PdfWriter.HideMenuBar	permet de spécifier si la barre de menu est affichée
PdfWriter.HideWindowUI	permet de spécifier si les contrôles de navigation sont affichés

Exemple :

```
package com.jmdoudoux.test.itext;

import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.PageSize;
import com.lowagie.text.Paragraph;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText4 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter writer = PdfWriter.getInstance(document,
          new FileOutputStream("c:/test.pdf"));
      writer.setViewerPreferences(PdfWriter.PageLayoutSinglePage
          | PdfWriter.PageModeUseThumbs);

      document.open();
      document.add(new Paragraph("Hello World"));
    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
```

```
      ioe.printStackTrace();
    }
    document.close();
  }
}
```



82.2.4. L'ajout de contenu au document

IText propose de nombreuses classes qui encapsulent des éléments qui pourront être ajoutés au contenu d'un document.

Cependant, toutes ces classes ne sont pas supportées par tous les DocWriters : si une classe n'est pas supportée par le DocWriter alors celle-ci est ignorée lors de l'exportation du document qui la contient.

82.2.4.1. Les polices de caractères

Par défaut, un document peut utiliser 14 polices de caractères standards : Courier, Courier Bold, Courier Italic, Courier Bold and Italic, Helvetica, Helvetica Bold, Helvetica Italic, Helvetica Bold and Italic, Times Roman, Times Roman Bold, Times Roman Italic, Times Roman Bold and Italic, Symbol et ZapfDingBats.

Chaque variante de type bold, italic et bold italic pour Courier, Helvetica et Times Roman sont proposées chacune sous la forme d'une police dédiée.

Une police de caractères est encapsulée dans un objet de type Font.

La classe Font encapsule les caractéristiques de la police de caractères : la famille, la taille, le style et la couleur. Elle possède de nombreux constructeurs pour définir ces différentes informations.

Elle propose des constantes pour :

- la famille : COURIER, HELVETICA, SYMBOL, TIMES_ROMAN, ZAPFDINGBATS
- la taille : DEFAULTSIZE
- le style : BOLD, BOLDITALIC, ITALIC, NORMAL, STRIKETHRU, UNDERLINE

Exemple :

```
package com.jmdoudoux.test.itext;
```

```
import java.awt.Color;
import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.Font;
import com.lowagie.text.PageSize;
import com.lowagie.text.Paragraph;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText5 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter writer = PdfWriter.getInstance(document,
          new FileOutputStream("c:/test.pdf"));
      document.open();
      document.add(new Paragraph("Hello World",
            new Font(Font.COURIER, 28, Font.BOLD, Color.RED)));
    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    document.close();
  }
}
```

iText propose une fabrique pour instancier des polices de caractères.

Exemple :

```
package com.jmdoudoux.test.itext;

import java.awt.Color;
import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.Font;
import com.lowagie.text.FontFactory;
import com.lowagie.text.PageSize;
import com.lowagie.text.Paragraph;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText6 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter.getInstance(document, new FileOutputStream("c:/test.pdf"));
      document.open();
      document.add(new Paragraph("Hello World", FontFactory.getFont(
          FontFactory.COURIER,
            28f,
            Font.BOLD,
            Color.RED)));
    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    document.close();
  }
```

```
}
```

Tous les objets retournés par la fabrique héritent de la classe BaseFont. La classe BaseFont propose plusieurs surcharges de la méthode createFont() pour instancier un objet de type Font.

Exemple :

```
package com.jmdoudoux.test.itext;

import java.awt.Color;
import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.Font;
import com.lowagie.text.PageSize;
import com.lowagie.text.Paragraph;
import com.lowagie.text.pdf.BaseFont;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText7 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter.getInstance(document, new FileOutputStream("c:/test.pdf"));
      document.open();

      BaseFont fonte = BaseFont.createFont(
          BaseFont.COURIER,
          BaseFont.CP1252,
          BaseFont.NOT_EMBEDDED);
      Font maFonte = new Font(fonte);
      maFonte.setColor(Color.RED);
      maFonte.setStyle(Font.BOLD);
      maFonte.setSize(38.0f);

      document.add(new Paragraph("Hello World", maFonte));
    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    document.close();
  }
}
```

Il est possible d'utiliser n'importe quelle fonte true type présente sur le système.

Exemple :

```
package com.jmdoudoux.test.itext;

import java.awt.Color;
import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.Font;
import com.lowagie.text.PageSize;
import com.lowagie.text.Paragraph;
import com.lowagie.text.pdf.BaseFont;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText8 {
```

```
  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter.getInstance(document, new FileOutputStream("c:/test.pdf"));
      document.open();

      BaseFont fonte = BaseFont.createFont(
          "C:/Windows/FONTS/ARIAL.TTF",
          BaseFont.CP1252,
          BaseFont.NOT_EMBEDDED);
      Font maFonte = new Font(fonte);
      maFonte.setColor(Color.RED);
      maFonte.setStyle(Font.BOLD);
      maFonte.setSize(38.0f);

      document.add(new Paragraph("Hello World", maFonte));
    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    document.close();
  }
}
```

La méthode createFont() possède plusieurs surcharges. Celle utilisée dans l'exemple attend en paramètre le chemin du fichier qui contient la police true type, l'encodage utilisé (plusieurs constantes sont définies : CP1250, CP1252, CP1257, MACROMAN, WINANSI) et un booléen qui précise si la police doit être incluse dans le PDF (deux constantes sont définies : EMBEDDED et NOT_EMBEDDED).

PDF fourni en standard 3 polices de caractères texte avec les styles normal, gras, italique et gras/italique (Courier, Helvetica et Times) et deux polices de symboles (Symbol et Zapf Dingbats). Il est donc inutile d'inclure ces polices dans le fichier PDF.

Il est possible d'utiliser la méthode register() de la classe FontFactory() pour enregistrer une police True Type en précisant le chemin du fichier de la police en paramètre. Une surcharge de cette méthode attend en plus en paramètre un nom d'alias pour accéder à la police.

Exemple :

```
package com.jmdoudoux.test.itext;

import java.awt.Color;
import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.Font;
import com.lowagie.text.FontFactory;
import com.lowagie.text.PageSize;
import com.lowagie.text.Paragraph;
import com.lowagie.text.pdf.BaseFont;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText9 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter.getInstance(document, new FileOutputStream("c:/test.pdf"));
      document.open();

      FontFactory.register("C:/Windows/FONTS/ARIAL.TTF");
```

```
      Font fonte = FontFactory.getFont("arial", BaseFont.WINANSI, 38);

      Font maFonte = new Font(fonte);
      maFonte.setColor(Color.RED);
      maFonte.setStyle(Font.BOLD);

      document.add(new Paragraph("Hello World", maFonte));
    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    document.close();
  }
}
```

82.2.4.2. Le classe Chunk

La classe Chunk encapsule une portion de texte du document affiché avec une certaine police de caractères. C'est la plus petite unité de texte utilisable.

Exemple :

```
package com.jmdoudoux.test.itext;

import java.awt.Color;
import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Chunk;
import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.Font;
import com.lowagie.text.FontFactory;
import com.lowagie.text.PageSize;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText10 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter.getInstance(document, new FileOutputStream("c:/test.pdf"));
      document.open();

      Chunk chunk = new Chunk("Hello world",
            FontFactory.getFont(FontFactory.COURIER, 20, Font.BOLD, Color.BLUE));

      document.add(chunk);

    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    document.close();
  }
}
```

La méthode setUnderline() permet pour les documents PDF d'avoir un contrôle précis sur les caractéristiques du soulignement de la portion de texte. Le premier paramètre précise l'épaisseur du trait et le second précise la position du trait.

Exemple :

```
package com.jmdoudoux.test.itext;

import java.awt.Color;
import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Chunk;
import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.Font;
import com.lowagie.text.FontFactory;
import com.lowagie.text.PageSize;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText11 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter.getInstance(document, new FileOutputStream("c:/test.pdf"));
      document.open();

      Chunk chunk = new Chunk("Hello world",
            FontFactory.getFont(FontFactory.COURIER, 20, Font.BOLD, Color.BLUE));
      chunk.setUnderline(0.2f,-2f);
      document.add(chunk);

      chunk = new Chunk("Hello world",
            FontFactory.getFont(FontFactory.COURIER, 20, Font.BOLD, Color.BLUE));
      chunk.setUnderline(2f,5f);
      document.add(chunk);

    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    document.close();
  }
}
```

Une surcharge de cette méthode permet de fournir des précisions sur l'apparence du trait.

Exemple :

```
package com.jmdoudoux.test.itext;

import java.awt.Color;
import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Chunk;
import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.Font;
import com.lowagie.text.FontFactory;
import com.lowagie.text.PageSize;
import com.lowagie.text.pdf.PdfContentByte;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText12 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter.getInstance(document, new FileOutputStream("c:/test.pdf"));
```

```
      document.open();

      Chunk chunk = new Chunk("Hello world",
            FontFactory.getFont(FontFactory.COURIER, 20, Font.BOLD, Color.BLUE));
      chunk.setUnderline(Color.BLUE, 5.0f, 0.0f, 0.0f, -0.2f,
            PdfContentByte.LINE_CAP_ROUND);
      document.add(chunk);

    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    document.close();
  }
}
```

La méthode setBackground() permet de modifier la couleur de fond de la portion de texte.

Exemple :

```
package com.jmdoudoux.test.itext;

import java.awt.Color;
import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Chunk;
import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.Font;
import com.lowagie.text.FontFactory;
import com.lowagie.text.PageSize;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText13 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter.getInstance(document, new FileOutputStream("c:/test.pdf"));
      document.open();

      Chunk chunk = new Chunk("Hello world",
            FontFactory.getFont(FontFactory.COURIER, 20, Font.BOLD, Color.WHITE));
      chunk.setBackground(Color.BLUE);
      document.add(chunk);


    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    document.close();
  }
}
```

82.2.4.3. La classe Phrase

La classe phrase encapsule une série d'objets de type Chunk qui définit une ou plusieurs lignes dont l'espacement est défini.

Elle possède de nombreux constructeurs.

Exemple :

```
package com.jmdoudoux.test.itext;

import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Chunk;
import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.PageSize;
import com.lowagie.text.Phrase;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText13 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter.getInstance(document, new FileOutputStream("c:/test.pdf"));
      document.open();

      Phrase phrase = new Phrase(new Chunk("Hello world "));
      phrase
          .add(new Chunk(
              " test de pharse dont la longueur dépasse largement une seule ligne"));
      phrase.add(new Chunk(" grace à un commentaire assez long"));
      document.add(phrase);

    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    document.close();
  }
}
```

La propriété leading de la classe Phrase permet de définir l'espacement entre deux lignes.

Exemple :

```
package com.jmdoudoux.test.itext;

import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Chunk;
import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.PageSize;
import com.lowagie.text.Phrase;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText14 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter.getInstance(document, new FileOutputStream("c:/test.pdf"));
      document.open();

      Phrase phrase = new Phrase(new Chunk("Hello world "));
      phrase.setLeading(20f);
      phrase
          .add(new Chunk(
```

```
              " test de pharse dont la longueur dépasse"+
                        " largement une seule ligne"));
      phrase.add(new Chunk(" grace à un commentaire assez long"));
      document.add(phrase);

    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    document.close();
  }
}
```

82.2.4.4. La classe Paragraph

La classe Paragrpah encapsule un ensemble d'objets de type Chunk et/ou Phrase pour former un paragraphe. Chacun de ces objets peut avoir des polices de caractères différents.

Un paragraphe commence systématiquement sur une nouvelle ligne.

La propriété leading permet de préciser l'espacement entre deux lignes.

Exemple :

```
package com.jmdoudoux.test.itext;

import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.PageSize;
import com.lowagie.text.Paragraph;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText15 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter.getInstance(document, new FileOutputStream("c:/test.pdf"));
      document.open();

      document.add(new Paragraph("ligne 1"));
      document.add(new Paragraph("ligne 2"));
      document.add(new Paragraph("ligne 3"));

    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    document.close();
  }
}
```

La méthode setAlignment() permet de définir l'alignement du paragraphe grâce à plusieurs constantes : Element.ALIGN_LEFT, Element.ALIGN_CENTER, Element.ALIGN_RIGHT et Element.ALIGN_JUSTIFIED

Exemple :

```
package com.jmdoudoux.test.itext;

import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.Element;
import com.lowagie.text.PageSize;
import com.lowagie.text.Paragraph;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText16 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter.getInstance(document, new FileOutputStream("c:/test.pdf"));
      document.open();

      document.add(new Paragraph("ligne 1"));
      Paragraph paragraph = new Paragraph("ligne 2");
      paragraph.setAlignment(Element.ALIGN_CENTER);
      document.add(paragraph);
      document.add(new Paragraph("ligne 3"));

    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    document.close();
  }
}
```

Il est possible de préciser une indentation à gauche et/ou droite respectivement grâce aux méthode setIndentationLeft() et setIndentationRight().

Exemple :

```
package com.jmdoudoux.test.itext;

import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.PageSize;
import com.lowagie.text.Paragraph;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText17 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter.getInstance(document, new FileOutputStream("c:/test.pdf"));
      document.open();

      Paragraph paragraph = new Paragraph(
          "ligne 1 test de phrase dont la longueur dépasse"+
                  " largement une seule ligne grace à un commentaire assez long");
      paragraph.setIndentationLeft(20f);
      document.add(paragraph);
      paragraph = new Paragraph(
          "ligne 2 test de phrase dont la longueur dépasse"+
                  " largement une seule ligne grace à un commentaire assez long");
      paragraph.setIndentationRight(20f);
```

```
      document.add(paragraph);

    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    document.close();
  }
}
```

Les méthodes setSpacingBefore() et setSpacingAfter() permettent respectivement de préciser l'espace avant et après le paragraphe.

82.2.4.5. La classe Chapter

La classe Chapter encapsule un chapitre. Elle hérite de la classe Section.

Un chapitre commence sur une nouvelle page et possède un numéro affiché par défaut.

Exemple :

```
package com.jmdoudoux.test.itext;

import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Chapter;
import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.PageSize;
import com.lowagie.text.Paragraph;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText18 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter.getInstance(document, new FileOutputStream("c:/test.pdf"));
      document.open();

      Chapter chapter = new Chapter(new Paragraph("Premier chapitre"), 1);

      Paragraph paragraph = new Paragraph("ligne 1 test de phrase");
      chapter.add(paragraph);
      paragraph = new Paragraph("ligne 2 test de phrase");
      chapter.add(paragraph);
      document.add(chapter);

      chapter = new Chapter(new Paragraph("Second chapitre"), 1);
      paragraph = new Paragraph("ligne 3 test de phrase");
      chapter.add(paragraph);
      document.add(chapter);

    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    document.close();
  }
}
```

Pour ne pas afficher le numéro, il faut invoquer la méthode setNumberDepth() avec la valeur 0 en paramètre.

82.2.4.6. La classe Section

La classe Section encapsule une section qui est un sous-ensemble d'un chapitre.

Pour ajouter une section, il faut utiliser la méthode addSection() qui retourne une instance de la Section.

Exemple :

```
package com.jmdoudoux.test.itext;

import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Chapter;
import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.PageSize;
import com.lowagie.text.Paragraph;
import com.lowagie.text.Section;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText19 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter.getInstance(document, new FileOutputStream("c:/test.pdf"));
      document.open();

      Chapter chapter = new Chapter(new Paragraph("Mon chapitre"), 1);

      Section section = chapter.addSection(new Paragraph("Premiere section "), 2);
      section.setChapterNumber(1);

      Paragraph paragraph = new Paragraph("ligne 1 test de phrase");
      section.add(paragraph);
      paragraph = new Paragraph("ligne 2 test de phrase");
      section.add(paragraph);

      section = chapter.addSection(new Paragraph("Seconde section "), 2);
      section.setChapterNumber(2);

      paragraph = new Paragraph("ligne 3 test de phrase");
      section.add(paragraph);

      document.add(chapter);

    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    document.close();
  }
}
```

La propriété numberdepth permet de préciser quelle est la profondeur de la section.

La méthode setChapterNumber() permet de préciser le numéro de la profondeur de la section.

82.2.4.7. La création d'une nouvelle page

Pour créer une nouvelle page, il faut invoquer la méthode newPage() de la classe Document.

Attention, l'appel à la méthode newPage() dans une page vide n'a aucun effet.

Pour créer une page vide, il faut ajouter un ligne

Exemple :

```
package com.jmdoudoux.test.itext;

import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Chapter;
import com.lowagie.text.Chunk;
import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.PageSize;
import com.lowagie.text.Paragraph;
import com.lowagie.text.Section;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText21 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter.getInstance(document, new FileOutputStream("c:/test.pdf"));
      document.open();

      Paragraph paragraph = new Paragraph("ligne 1 test de phrase");
      document.add(paragraph);
      document.newPage();
      document.add(Chunk.NEWLINE);
      document.newPage();

      paragraph = new Paragraph("ligne 2 test de phrase");
      document.add(paragraph);
    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    document.close();
  }
}
```

Dans une section, il faut lui ajouter un objet de type Chunk.NEXTPAGE.

Exemple :

```
package com.jmdoudoux.test.itext;

import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Chapter;
import com.lowagie.text.Chunk;
import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.PageSize;
import com.lowagie.text.Paragraph;
import com.lowagie.text.Section;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText20 {
```

```
  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter.getInstance(document, new FileOutputStream("c:/test.pdf"));
      document.open();

      Chapter chapter = new Chapter(new Paragraph("Mon chapitre"), 1);

      Section section = chapter.addSection(new Paragraph("Premiere section "), 2);
      section.setChapterNumber(1);

      Paragraph paragraph = new Paragraph("ligne 1 test de phrase");
      section.add(paragraph);
      paragraph = new Paragraph("ligne 2 test de phrase");
      section.add(paragraph);
      section.add(Chunk.NEXTPAGE);

      section = chapter.addSection(new Paragraph("Seconde section "), 2);
      section.setChapterNumber(2);

      paragraph = new Paragraph("ligne 3 test de phrase");
      section.add(paragraph);

      document.add(chapter);
    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    document.close();
  }
}
```

82.2.4.8. La classe Anchor

La classe Anchor encapsule un lien hypertexte. Elle hérite de la classe Phrase.

La méthode setReference() permet de préciser l'url externe du lien.

La méthode setName() permet de préciser l'ancre pour un lien interne.

Exemple :

```
package com.jmdoudoux.test.itext;

import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Anchor;
import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.PageSize;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText22 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter.getInstance(document, new FileOutputStream("c:/test.pdf"));
      document.open();

      Anchor anchor = new Anchor("mon site web");
      anchor.setReference("http://www.jmdoudoux.fr/");
```

```
      document.add(anchor);

    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    document.close();
  }
}
```

82.2.4.9. Les classes List et ListItem

La classe List encapsule une liste d'éléments de type ListeItem.

La classe Liste possède plusieurs constructeurs. La liste peut être ordonnée ou non selon le premier paramètre fourni au constructeur utilisé : true indique une liste ordonnée.

Exemple :

```
package com.jmdoudoux.test.itext;

import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.List;
import com.lowagie.text.ListItem;
import com.lowagie.text.PageSize;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText23 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter.getInstance(document, new FileOutputStream("c:/test.pdf"));
      document.open();

      List liste = new List(true, 20);
      liste.add(new ListItem("Element 1"));
      liste.add(new ListItem("Element 2"));
      liste.add(new ListItem("Element 3"));
      document.add(liste);

      liste = new List(false, 30);
      liste.add(new ListItem("Element 1"));
      liste.add(new ListItem("Element 2"));
      liste.add(new ListItem("Element 3"));
      document.add(liste);

    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    document.close();
  }
}
```

La méthode setLettered() permet de préciser si la numérotation d'une liste ordonnée est littérale : la première valeur est dans ce cas A.

La méthode setNumbered() permet de préciser si la numérotation d'une liste ordonnée est numérique : la première valeur est dans ce cas 1.

Dans une liste ordonnée, la méthode setFirst() permet de préciser la valeur du premier élément pour une numérotation numérique ou littérale.

Dans une liste non ordonnée, la méthode setListeSymbol() permet de préciser quels seront le ou les caractères utilisés comme puce.

Exemple :

```
package com.jmdoudoux.test.itext;

import java.awt.Color;
import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Chunk;
import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.Font;
import com.lowagie.text.FontFactory;
import com.lowagie.text.List;
import com.lowagie.text.ListItem;
import com.lowagie.text.PageSize;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText24 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter.getInstance(document, new FileOutputStream("c:/test.pdf"));
      document.open();

      List liste = new List(20);
      liste.setListSymbol(new Chunk("B",
            FontFactory.getFont(FontFactory.ZAPFDINGBATS, 20, Font.BOLD, Color.BLUE)));
      liste.add(new ListItem("Element 1"));
      liste.add(new ListItem("Element 2"));
      liste.add(new ListItem("Element 3"));
      document.add(liste);

    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    document.close();
  }
}
```

82.2.4.10. La classe Table

La classe com.lowagie.test.Table encapsule un tableau utilisé comme une matrice. Chaque cellule est encapsulée dans un objet de type Cell.

Il est possible de définir le nombre de lignes et de colonnes en utilisant la surcharge du constructeur adéquat.

Il est impératif de définir le nombre de colonnes ; le nombre de lignes peut croître selon les besoins.

La méthode addCell() permet de fournir la valeur de la cellule courante. Par défaut, c'est la cellule de la première ligne, première colonne. L'appel à la méthode déplace la cellule courante dans la même ligne sur la colonne suivante si elle existe sinon sur la première colonne de la ligne suivante.

Exemple :

```
package com.jmdoudoux.test.itext;

import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.PageSize;
import com.lowagie.text.Table;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText25 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter.getInstance(document, new FileOutputStream("c:/test.pdf"));
      document.open();

      Table tableau = new Table(2,2);
      tableau.addCell("1.1");
      tableau.addCell("1.2");
      tableau.addCell("2.1");
      tableau.addCell("2.2");

      document.add(tableau);


    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    document.close();
  }
}
```

La classe Table possède une surcharge de la méthode addCell() qui attend en second paramètre un objet de type Point qui permet d'indiquer une cellule bien précise dans le tableau.

La méthode setAutoFillEmptyCell() attend un booléen qui permet de préciser si les cellules non renseignées doivent être automatiquement créées vides.

Exemple :

```
package com.jmdoudoux.test.itext;

import java.awt.Point;
import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.PageSize;
import com.lowagie.text.Table;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText26 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
```

```
    try {
      PdfWriter.getInstance(document, new FileOutputStream("c:/test.pdf"));
      document.open();

      Table tableau = new Table(2,2);
      tableau.addCell("1.0", new Point(1,0));
      tableau.addCell("2.1", new Point(2,1));

      document.add(tableau);


    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    document.close();
  }
}
```

La classe Table possède de nombreuses méthodes pour modifier son rendu par exemple : setBorderWidth(), setBorderColor(), setBackGroundColor(), setPadding(), ...

Pour obtenir plus de souplesse dans le rendu d'une cellule, il est possible d'instancier une occurrence de la classe Cell et d'invoquer les méthodes qu'elle propose pour configurer son apparence.

Exemple :

```
package com.jmdoudoux.test.itext;

import java.awt.Color;
import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Cell;
import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.Element;
import com.lowagie.text.PageSize;
import com.lowagie.text.Table;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText27 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter.getInstance(document, new FileOutputStream("c:/test.pdf"));
      document.open();

      Table tableau = new Table(2,2);
      tableau.setAutoFillEmptyCells(true);
      tableau.setPadding(2);

      Cell cell = new Cell("1.1");
      cell.setHorizontalAlignment(Element.ALIGN_CENTER);
      cell.setBackgroundColor(Color.YELLOW);
      tableau.addCell(cell);

      tableau.addCell("1.2");
      tableau.addCell("2.1");
      tableau.addCell("2.2");

      document.add(tableau);

    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
```

```
    }

    document.close();
  }
}
```

Il est possible de définir une en-tête pour les colonnes en ajoutant au début des cellules un appel à la méthode setHeader() avec le paramètre true. Une fois toutes les en-têtes définies, il faut invoquer la méthode endHeaders() de la classe Table.

Exemple :

```
package com.jmdoudoux.test.itext;

import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Cell;
import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.Element;
import com.lowagie.text.PageSize;
import com.lowagie.text.Table;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText28 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter.getInstance(document, new FileOutputStream("c:/test.pdf"));
      document.open();

      Table tableau = new Table(2, 2);
      tableau.setAutoFillEmptyCells(true);
      tableau.setPadding(2);

      Cell cell = new Cell("colonne 1");
      cell.setHeader(true);
      cell.setHorizontalAlignment(Element.ALIGN_CENTER);
      tableau.addCell(cell);

      cell = new Cell("colonne 2");
      cell.setHeader(true);
      cell.setHorizontalAlignment(Element.ALIGN_CENTER);
      tableau.addCell(cell);
      tableau.endHeaders();

      cell = new Cell("1.1");
      cell.setHorizontalAlignment(Element.ALIGN_CENTER);
      tableau.addCell(cell);

      tableau.addCell("1.2");
      tableau.addCell("2.1");
      tableau.addCell("2.2");

      document.add(tableau);

    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    document.close();
  }
}
```

82.2.5. Des fonctionnalités avancées

iText propose de nombreuses fonctionnalités avancées pour générer un document.

82.2.5.1. Insérer une image

iText propose le support de plusieurs formats d'images : JPEG, GIF, PNG, BMP, TIFF, WMF et les objets de type java.awt.image

La classe Image est une classe abstraite dont hérite chaque classe qui encapsule un type d'images supporté par iText.

La méthode getInstance() de la classe Image permet d'obtenir une instance d'une image. De nombreuses surcharges sont proposées pour fournir par exemple un chemin sur le système de fichiers ou une url.

Exemple :

```
package com.jmdoudoux.test.itext;

import java.io.FileOutputStream;
import java.io.IOException;

import com.lowagie.text.Document;
import com.lowagie.text.DocumentException;
import com.lowagie.text.Image;
import com.lowagie.text.PageSize;
import com.lowagie.text.pdf.PdfWriter;

public class TestIText29 {

  public static void main(String[] args) {

    Document document = new Document(PageSize.A4);
    try {
      PdfWriter.getInstance(document, new FileOutputStream("c:/test.pdf"));
      document.open();

      Image image = Image.getInstance("c:/monimage.jpg");
      document.add(image);

    } catch (DocumentException de) {
      de.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    document.close();
  }
}
```

La méthode setAlignment() peut être utilisée pour préciser l'alignement de l'image en passant en paramètre une des constantes : LEFT, MIDDLE ou RIGHT

La classe Image propose plusieurs méthodes pour permettre de redimentionner l'image : scaleAbsolute(), scaleAbsoluteWidth(), scaleAbsoluteHeight(), scalePercent() et scaleToFit().

Les méthodes setRotation() et setRotationDegrees() permettent de faire une rotation de l'image.

83. La validation des données

La validation des données est une tâche commune, nécessaire et importante dans chaque application. De plus, ces validations peuvent être faites dans les différentes couches d'une application :

- Présentation
- Service
- Métier
- DAO
- Dans la base de données via des contraintes d'intégrités

Certains frameworks notamment pour les couches présentation et DAO propose leur propre solution de validation de données. Pour les autres couches, soit un autre framework soit une solution maison sont utilisées. Toutes ces solutions proposent des implémentations différentes pour déclarer et valider des contraintes mais aussi pour signaler les violations de contraintes (Exception, objets dédiés, ...).

Ceci entraine fréquemment une duplication du code et/ou une redondance des contrôles effectuées avec les risques que cela peut engendrer :

- temps requis
- source d'erreurs
- difficultés de maintenance

Il y a aussi le risque d'oublier la déclaration de contraintes dans une couche.

Une solution est de mettre ces traitements de validation dans les entités du domaine ce qui les complexifient.

De plus, certaines de ces validations sont fréquemment utilisées et sont donc standards (vérifier la présence d'une valeur, vérifier une taille, vérifier la valeur sur une plage de date ou une plage numérique, vérifier la valeur sur une expression régulière, ...).

Il est aussi généralement nécessaire de développer des validations spécifiques.

Pour répondre à ces différents besoins, des frameworks ont été développés pour :

- fournir des valideurs classiques
- permettre de définir ces propres valideurs
- faciliter l'application de ces valideurs sur des données.

Ce chapitre contient plusieurs sections :

♦ Quelques recommandations sur la validation des données
♦ L'API Bean Validation (JSR 303)
♦ D'autres frameworks pour la validation des données

83.1. Quelques recommandations sur la validation des données

Les validations de données sont utiles dans plusieurs endroits d'une application et ce quel que soit le type d'applications :

- Couche présentation : pour informer le plus rapidement possible d'une donnée erronée à l'utilisateur, généralement se sont des contrôles de surface
- Couche service : validation des données reçues et qui n'ont pas été forcément validées par une IHM
- Couche métier : validation des données traitées qui peuvent nécessiter une accès aux données
- Couche accès aux données (DAO) : validation des données avant leur envoie dans la base de données

Il est important de valider les contraintes le plus tôt possible dans les couches de l'application pour éviter des appels inutiles, fréquemment distants aux fonctionnalités de la couche en amont.

Il est aussi très important de répéter les validations de données dans les couches sous jacentes car il ne faut pas présumer de ce que fait la couche en amont. Par exemple, même si les données dont validées dans l'IHM d'une application invoquant un service, il faut revalider ces données dans la couche service car si le service est invoquée par une application qui ne fait pas le contrôle, les données risque d'être non valides. Même si cela augmente les traitements cela rend les traitements plus sûres.

Dans tous les cas, les contrôles doivent être faits dans la couche plus basse possible, incluant dans la base de données via des contrôles d'intégrité.

Il existe des contrôles spécifiques à la couche présentation : par exemple, la double saisie d'un mot de passe et la comparaison des deux valeurs saisies.

Il existe une frontière très mince entre les règles métiers, les traitements métiers et la validation des données. Il peut être tentant de mettre certaines de ces fonctionnalités dans la validation des données mais il ne faut pas tout mettre dans la validation et maintenir un rôle à la couche métier. Les traitements de validation des données doivent rester simple et ne doivent pas devenir trop complexes ni nécessiter plusieurs entités (propriétés, objets, ressources, ...).

83.2. L'API Bean Validation (JSR 303)

L'API Bean Validation est issue des travaux de la JSR 303 : http://jcp.org/en/jsr/detail?id=303

Cette JSR 303 propose de standardiser un framework de validation des données d'un bean.

L'intérêt de cette API est de proposer une approche cohérente sous la forme d'un standard pour la validation des données d'un bean.

Il y a besoin d'un standard pour plusieurs raisons :

- Il existe déjà plusieurs frameworks open source de validation de données
- Les validations se font dans toutes les couches, pas toujours de façon cohérente et fréquemment par duplication de code
- Plusieurs technologies des plateformes Java ont besoin d'un framework de validation : JPA, JSF, ...

Généralement ces validations ont lieu avec plus ou moins de redondance dans les différentes couches d'une application.

Fréquemment, les contraintes sont exprimées sur les entités du domaine ainsi la JSR propose de déclarer les contraintes dans les beans qui encapsulent les entités du domaine.

Inclure ces validations dans les entités du domaine permet de centraliser ces traitements plutôt que de les dupliquer ou les répartir dans les différentes couches.

83.2.1. La présentation de l'API

L'API Bean Validation standardise la définition, la déclaration et la validation de contraintes sur les données d'un ou plusieurs beans.

La déclaration de contraintes se fait dans le bean qui encapsule les données. L'expression de ces contraintes se fait à l'aide d'annotations ou d'un descripteur au format XML ce qui permet de réduire la quantité de code à produire. La manière privilégiée pour déclarer les contraintes est d'utiliser les annotations mais il est aussi possible d'utiliser un descripteur au format XML.

L'API propose une ensemble de contraintes communes fournies en standard et permet de définir ces propres contraintes.

La validation de ces contraintes se fait grâce à un valideur fourni par l'API.

Elle propose aussi des fonctionnalités avancées comme la composition de contraintes, la validation partielle en utilisant la notion de groupes de contraintes, la définition de contraintes personnalisées et la recherche des contraintes définies.

83.2.1.1. Les objectifs de l'API

La JSR 303 tente de combiner les meilleures fonctionnalités de différents frameworks dans une spécification qui peut être implémentée par différents fournisseurs et qui propose :

- De fournir un ensemble de contraintes standards
- De déclarer les contraintes sans avoir à écrire de code explicitement
- De valider un objet par rapport à ses contraintes et à ses valeurs grâce à un moteur de validation
- De définir une contrainte personnalisée pour rendre le framework extensible
- De fournir une API pour rechercher les contraintes sur un type

Le but de cette JSR est de standardiser les fonctionnalités de validation des données des beans en utilisant des annotations plutôt que d'avoir à écrire du code pour réaliser ces validations.

Le but n'est pas de fournir une solution permettant de définir des contraintes dans tous les tiers (notamment elle ne couvre pas directement les contraintes dans la base de données car celles-ci sont spécifiques) mais propose de définir les contraintes au niveau des entités du domaine. Ce choix repose sur le fait que ces contraintes sont généralement liées à l'entité elle-même.

La JSR 303 a plusieurs objectifs :

- Proposer une spécification relative à la validation de données dans les applications Java
- Fournir une API qui soit indépendante d'une architecture
- Etre utilisable dans toutes les couches Java d'une application : elle est utilisable côté client ou serveur
- Standardiser la déclaration des contraintes en privilégiant le annotations au niveau de la classe (généralement un bean qui encapsule une entité du domaine)
- Définir des contraintes communes fournies en standard
- Fournir un mécanisme standard pour valider les contraintes
- Etre facile à utiliser et extensible
- Fournir une API qui permet de rechercher les contraintes exploitable notamment par des frameworks

83.2.1.2. Les éléments et concepts utilisés par l'API

L'API Java Bean Validation utilise plusieurs éléments et concepts lors de sa mise en oeuvre :

- Une contrainte est une restriction appliquée sur la valeur d'un champ ou d'une propriété d'une instance d'un bean.
- La déclaration d'une contrainte assigne une contrainte à un bean, un champ ou une propriété en utilisant une annotation ou grâce à un fichier XML.
- Une implémentation de l'interface ConstraintValidator encapsule les traitements de validation d'une donnée ou du bean.

La définition d'une contrainte se fait via une annotation en précisant le type sur lequel elle s'applique, ses attributs et la classe qui encapsule les traitements de validation.

Les groupes permettent de n'appliquer qu'un sous-ensemble des contraintes d'un bean. La déclaration d'une contrainte peut être associée à un ou plusieurs groupes.

La validation d'une contrainte applique les traitements de validation d'une données sur l'instance courante.

Les spécifications doivent être implémentées par un fournisseur pour pouvoir être utilisées. Le projet Hibernate Validator est l'implémentation de référence de ces spécifications.

L'interpolation du message contient les traitements pour créer le message d'erreur fourni à l'utilisateur.

Une séquence permet de définir l'ordre dans lequel les contraintes de validation vont être évaluées.

Les spécifications proposent une API qui permet d'obtenir des métadonnées sur les contraintes d'un type. Cette API est particulièrement utile pour l'intégration dans d'autres frameworks.

Les spécifications proposent aussi une API nommée BootStrap qui fournit des mécanismes pour obtenir une instance de la fabrique de type ValidatorFactory. Cette API permet notamment de choisir l'implémentation à utiliser.

83.2.1.3. Les contraintes et leur validation avec l'API

Une contrainte est composée de deux éléments :

- Une annotation : utilisée par le développeur pour déclarer ses contraintes
- Une classe de type Validator : utilisée par l'API pour valider les données selon les annotations utilisées

La JSR-303 définit un ensemble de contraintes que chaque implémentation doit fournir. Cependant, cet ensemble ne concerne que des contraintes standards qui ne sont en général pas suffisantes pour répondre à tous les besoins. La spécification prévoit donc la possibilité de développer ces propres contraintes personnalisées.

La définition de ces propres contraintes peut se faire de deux façons :

- Par combinaison d'autres contraintes sous la forme d'une composition de contraintes qui est un ensemble de contraintes qui seront utilisées comme une seule
- Par la définition de ses propres contraintes

La validation des contraintes se fait par introspection à la recherche des annotations du type des contraintes utilisées dans le bean. Pour chaque annotation, la classe de type Validator associée est instanciée et utilisée par le framework pour valider la valeur de la donnée.

L'API peut prendre en charge, à la demande lors de la validation du bean, le parcours des objets dépendants du bean à valider pour les valider aussi si ceux-ci possèdent des contraintes définies.

La validation des données peut être invoquée automatiquement par les frameworks qui proposent un support pour l'API Bean Validator : c'est notamment le cas pour JSF 2.0 et JPA 2.0.

83.2.1.4. La mise en oeuvre générale de l'API

La JSR 303 propose de standardiser les validations via les spécifications d'une API composée de plusieurs parties :

- des annotations sur les entités du domaine ce qui permet de centraliser ces validations sans alourdir les beans avec beaucoup de code
- une API pour valider les contraintes
- une API dédiée à l'obtention des méta données des contraintes

L'API est conçue pour être utilisée avec n'importe quel bean dans n'importe quelle couche écrite en Java d'une application.

La plupart des frameworks de validations sont relatifs à un framework particulier pour une ou deux couches données : Struts, Hibernate, ... L'API Bean Validator est conçue pour permettre une utilisation dans toutes les couches écrites en Java d'une application.

L'API Bean Validation est incluse dans Java EE 6 car elle est utilisée par JSF 2.0 et JPA 2.0. L'API peut cependant être utilisée dans Java SE à partir de la version 5.

L'API Bean Validation est conçue pour être indépendante de la technologie qui l'utilise aussi bien côté client (Swing, ...) que serveur (JPA, JSF, ...).

Le package de cette API est javax.validation.

L'implémentation de référence est proposée par Hibernate Validator 4.

Pour mettre en oeuvre l'API, il n'est pas nécessaire d'utiliser des classes de l'implémentation : seules les classes et interfaces de l'API doivent être importées dans le code source. Ceci rend l'utilisation d'une autre implémentation très facile.

Il est nécessaire d'ajouter au classpath les dépendances de l'implémentation utilisée : par exemple, avec l'implémentation de référence.

83.2.1.5. Un exemple simple de mise en oeuvre

Cet exemple va définir un bean, ajouter une contrainte de type non null sur un champ et créer une petite application de test qui va instancier le bean avec un champ null et appliquer les validations des contraintes sur le bean.

La JSR 303 permet d'annoter une classe ou un attribut d'une classe ou le getter de cet attribut.

Exemple :

```
package com.jmdoudoux.test.validation;

import java.util.Date;

import javax.validation.constraints.NotNull;
import javax.validation.constraints.Past;
import javax.validation.constraints.Size;

public class PersonneBean {

  private String nom;
  private String Prenom;
  private Date dateNaissance;

  public PersonneBean(String nom, String prenom, Date dateNaissance) {
    super();
    this.nom = nom;
    Prenom = prenom;
    this.dateNaissance = dateNaissance;
  }

  @NotNull
  @Size(max=50)
  public String getNom() {
    return nom;
  }

  public void setNom(String nom) {
    this.nom = nom;
  }

  @NotNull
```

```
  @Size(max=50)
  public String getPrenom() {
    return Prenom;
  }

  public void setPrenom(String prenom) {
    Prenom = prenom;
  }

  @Past
  public Date getDateNaissance() {
    return dateNaissance;
  }

  public void setDateNaissance(Date dateNaissance) {
    this.dateNaissance = dateNaissance;
  }

}
```

L'API propose aussi un mécanisme pour valider les contraintes et exploiter les éventuelles violations.

Exemple :

```
package com.jmdoudoux.test.validation;

import java.util.Calendar;
import java.util.GregorianCalendar;
import java.util.Set;

import javax.validation.ConstraintViolation;
import javax.validation.Validation;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;

public class TestValidation {

  public static void main(String[] args) {

    PersonneBean personne = new PersonneBean(null, null, new GregorianCalendar(
        2065, Calendar.JANUARY, 18).getTime());

    ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
    Validator validator = factory.getValidator();

    Set<ConstraintViolation<PersonneBean>> constraintViolations =
      validator.validate(personne);

    if (constraintViolations.size() > 0 ) {
      System.out.println("Impossible de valider les donnees du bean : ");
      for (ConstraintViolation<PersonneBean> contraintes : constraintViolations) {
        System.out.println(contraintes.getRootBeanClass().getSimpleName()+
           "." + contraintes.getPropertyPath() + " " + contraintes.getMessage());
      }
    } else {
      System.out.println("Les donnees du bean sont valides");
    }
  }
}
```

Résultat :

```
Impossible de valider les donnees du bean :
PersonneBean.dateNaissance doit être dans le passé
PersonneBean.nom ne peut pas être nul
PersonneBean.prenom ne peut pas être nul
```

83.2.2. La déclaration des contraintes

La déclaration de contraintes se fait dans des classes ou des interfaces avec des annotations qui est la manière recommandée ou via une description dans un fichier XML.

Une contrainte peut être appliquée sur un type (classe ou interface), un champ ou une propriété respectant les conventions des Java beans.

Remarque : les champs statiques ne peuvent pas être validés en utilisant l'API.

La valeur fournie à l'objet de type ConstraintValidator qui va valider les contraintes dépend de l'entité annotée avec la contrainte :

- Si la contrainte est définie sur la classe ou une interface de la classe alors c'est l'instance de classe qui est fournie comme valeur
- Si la contrainte est définie sur un champ alors c'est la valeur du champ qui est fournie
- Si la contrainte est définie sur le getter d'une propriété alors c'est la valeur de retour de ce getter qui est fournie

Remarque : il faut définir les contraintes soit sur-le-champ soit sur la propriété correspondante mais pas sur les deux à la fois sinon la validation se fera deux fois. Il est préférable de rester consistant et d'utiliser les annotations toujours sur les champs ou toujours sur les getter.

Chaque déclaration d'une contrainte peut redéfinir le message fourni en cas de violation de la contrainte.

83.2.2.1. La déclaration des contraintes sur les champs

L'application de contraintes sur un champ permet de réaliser la validation de la donnée par l'implémentation de l'API de façon indépendante par rapport à la forme d'accès faite à ce champ par l'API.

Exemple :

```
package com.jmdoudoux.test.validation;

import java.util.Date;

import javax.validation.constraints.NotNull;

public class PersonneBean {

  @NotNull
  private String nom;
  @NotNull
  private String Prenom;
  @Past
  private Date dateNaissance;

  public PersonneBean(String nom, String prenom, Date dateNaissance) {
    super();
    this.nom = nom;
    Prenom = prenom;
    this.dateNaissance = dateNaissance;
  }

  public String getNom() {
    return nom;
  }

  public void setNom(String nom) {
    this.nom = nom;
  }

  public String getPrenom() {
    return Prenom;
  }
```

```
  public void setPrenom(String prenom) {
    Prenom = prenom;
  }

  public Date getDateNaissance() {
    return dateNaissance;
  }

  public void setDateNaissance(Date dateNaissance) {
    this.dateNaissance = dateNaissance;
  }

}
```

L'application de ces contraintes peut se faire sur un champ quel que soit sa visibilité (private, protected ou public) mais ne peut pas se faire sur un champ static.

83.2.2.2. La déclaration des contraintes sur les propriétés

Il est possible de définir les contraintes sur une propriété : dans ce cas, seul le getter doit être annoté.

Exemple :

```
package com.jmdoudoux.test.validation;

import java.util.Date;

import javax.validation.constraints.NotNull;

public class PersonneBean {

  private String nom;
  private String Prenom;
  private Date dateNaissance;

  public PersonneBean(String nom, String prenom, Date dateNaissance) {
    super();
    this.nom = nom;
    Prenom = prenom;
    this.dateNaissance = dateNaissance;
  }

  @NotNull
  public String getNom() {
    return nom;
  }

  public void setNom(String nom) {
    this.nom = nom;
  }

  @NotNull
  public String getPrenom() {
    return Prenom;
  }

  public void setPrenom(String prenom) {
    Prenom = prenom;
  }

  @Past
  public Date getDateNaissance() {
    return dateNaissance;
  }

  public void setDateNaissance(Date dateNaissance) {
    this.dateNaissance = dateNaissance;
  }
```

```
}
```

La validation de la donnée par l'implémentation de l'API utilise alors obligatoirement le getter pour obtenir la valeur de la donnée.

83.2.2.3. La déclaration des contraintes sur une classe

La déclaration d'une contrainte peut être faite sur une classe ou une interface. Dans ce cas, la validation se fait sur l'état de la classe ou de la classe qui implémente l'interface.

C'est instance de la classe qui sera fournie comme valeur à valider au ConstraintValidator.

Une telle validation peut être requise si elle nécessite l'état de plusieurs données de la classe pour être réalisée.

83.2.2.4. L'héritage de contraintes

Lorsqu'un bean hérite d'un autre bean qui contient une définition de contraintes, celles-ci sont héritées.

Exemple :

```
package com.jmdoudoux.test.validation;

import java.util.Date;

import javax.validation.constraints.Min;

public class DeveloppeurSeniorBean extends PersonneBean {

  private int experience;

  public DeveloppeurSeniorBean(String nom, String prenom, Date dateNaissance, int experience) {
    super(nom, prenom, dateNaissance);
    this.experience = experience;
  }

  @Min(value=5)
  public int getExperience() {
    return experience;
  }

  public void setExperience(int experience) {
    this.experience = experience;
  }

}
```

Lors de la validation du bean, les contraintes du bean sont vérifiées mais aussi celle de la classe mère.

Exemple :

```
package com.jmdoudoux.test.validation;

import java.util.Calendar;
import java.util.GregorianCalendar;
import java.util.Set;

import javax.validation.ConstraintViolation;
import javax.validation.Validation;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;
```

```
public class TestValidation {

  public static void main(String[] args) {
    DeveloppeurSeniorBean personne = new DeveloppeurSeniorBean(null, "", new GregorianCalendar(
        1965, Calendar.JANUARY, 18).getTime(), 3);

    ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
    Validator validator = factory.getValidator();

    Set<ConstraintViolation<DeveloppeurSeniorBean>> constraintViolations =
      validator.validate(personne);

    if (constraintViolations.size() > 0 ) {
      System.out.println("Impossible de valider les donnees du bean : ");
      for (ConstraintViolation<DeveloppeurSeniorBean> contraintes : constraintViolations) {
        System.out.println(contraintes.getRootBeanClass().getSimpleName()+
          "." + contraintes.getPropertyPath() + " " + contraintes.getMessage());
      }
    } else {
      System.out.println("Les donnees du bean sont valides");
    }
  }
}
```

Résultat :

```
Impossible de valider les donnees du bean :
DeveloppeurSeniorBean.experience doit être plus grand que 5
DeveloppeurSeniorBean.nom ne peut pas être nul
```

Les contraintes sont héritées d'une classe mère mais elles peuvent être redéfinies.

Si une méthode est redéfinie, les contraintes de la méthode de la classe mère s'appliquent aussi sauf si une contrainte existante est aussi redéfinie.

83.2.2.5. Les contraintes de validation d'un ensemble d'objets

L'API propose une validation d'un objet mais permet aussi la validation d'un graphe d'objets composés de l'objet et de tout ou partie de ses objets dépendants.

L'annotation @Valid utilisée sur une dépendance d'un bean permet de demander au moteur de validation de valider aussi la dépendance lors de la validation du bean.

Ce mécanisme est récursif : une dépendance annotée avec @Valid peut elle-même contenir des dépendances annotées avec @Valid. Ainsi, l'ensemble des beans dépendants qui seront validés en même temps que le bean est défini en utilisant l'annotation @Valid sur chacune des dépendances concernées.

Une dépendance annotée avec @Valid est ignorée par le moteur si sa valeur est nulle.

Exemple :

```
package com.jmdoudoux.test.validation;

import javax.validation.Valid;
import javax.validation.constraints.NotNull;

public class Comite {

  @NotNull
  @Valid
  private PersonneBean president;

  @Valid
  private PersonneBean tresorier;
```

```
  @Valid
  private PersonneBean secretaire;

  public Comite(PersonneBean president, PersonneBean tresorier,
      PersonneBean secretaire) {
    super();
    this.president = president;
    this.tresorier = tresorier;
    this.secretaire = secretaire;
  }

  public PersonneBean getPresident() {
    return president;
  }

  public PersonneBean getTresorier() {
    return tresorier;
  }

  public PersonneBean getSecretaire() {
    return secretaire;
  }
}
```

La validation du bean échoue si la validation d'une de ces dépendances échoue.

La dépendance peut aussi être une collection typée de beans. Cette collection peut être :

- un tableau
- une implémentation de java.lang.Iterable (collection, List, Set, ...)
- une implémentation de java.util.Map

Exemple :

```
package com.jmdoudoux.test.validation;

import java.util.ArrayList;
import java.util.List;

import javax.validation.Valid;
import javax.validation.constraints.NotNull;

public class Groupe {

  @NotNull
  private String nom;

  private List<PersonneBean> membres = new ArrayList<PersonneBean>();

  public Groupe(String nom) {
    super();
    this.nom = nom;
    membres = new ArrayList<PersonneBean>();
  }

  public String getNom() {
    return nom;
  }

  @NotNull
  @Valid
  public List<PersonneBean> getMembres() {
    return membres;
  }

  public void ajouter(PersonneBean personne) {
    membres.add(personne);
  }
```

```
  public void supprimer(PersonneBean personne) {
    membres.remove(personne);
  }
}
```

Si une telle collection est marquée avec l'annotation @Valid, alors toutes les occurrences de la collection seront validées lorsque le bean qui encapsule la collection sera validé.

Exemple :

```
package com.jmdoudoux.test.validation;

import java.util.Calendar;
import java.util.GregorianCalendar;
import java.util.Set;

import javax.validation.ConstraintViolation;
import javax.validation.Validation;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;

public class TestValidationGroupe {

  public static void main(String[] args) {


    Groupe groupe = new Groupe("Mon groupe");

    PersonneBean personne = new PersonneBean(null, null, new GregorianCalendar(
        2065, Calendar.JANUARY, 18).getTime());

    groupe.ajouter(personne);

    ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
    Validator validator = factory.getValidator();

    Set<ConstraintViolation<Groupe>> constraintViolations =
      validator.validate(groupe);

    if (constraintViolations.size() > 0 ) {
      System.out.println("Impossible de valider les donnees du bean : ");
      for (ConstraintViolation<Groupe> contraintes : constraintViolations) {
        System.out.println(contraintes.getRootBeanClass().getSimpleName()+
          "." + contraintes.getPropertyPath() + " " + contraintes.getMessage());
      }
    } else {
      System.out.println("Les donnees du groupe sont valides");
    }

  }
}
```

Résultat :

```
Impossible de valider les donnees du bean :
Groupe.membres[0].nom ne peut pas être nul
Groupe.membres[0].dateNaissance doit être dans le passé
Groupe.membres[0].prenom ne peut pas être nul
```

Les occurrences null dans une collection sont ignorées lors de la validation.

Dans le cas d'une collection de type Map, seules les valeurs sont validées (Map.Entry) : les clés ne sont pas validées.

Lors de la validation, l'annotation @Valid est traitée récursivement dans les dépendances tant que cela ne provoque pas une boucle infinie : le moteur de validation doit ignorer une instance qui a déjà été validée lors du traitement d'un même graphe d'objets.

83.2.3. La validation des contraintes

Bean Validation propose une API pour permettre la validation des contraintes sur les données de façon indépendante du tiers dans lequel elle est mise en oeuvre.

L'interface Validator définit les fonctionnalités d'un valideur.

Pour valider les contraintes sur les données d'un bean, il faut obtenir une instance de l'interface Validator, utiliser cette instance pour valider les données d'un bean. Les éventuelles erreurs détectées par cette validation sont retournées sous la forme d'une collection de type Set d'objets de type ConstraintViolation.

Exemple :

```
package com.jmdoudoux.test.validation;

import java.util.Calendar;
import java.util.GregorianCalendar;
import java.util.Set;

import javax.validation.ConstraintViolation;
import javax.validation.Validation;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;

public class TestValidation {

  public static void main(String[] args) {

    PersonneBean personne = new PersonneBean("nom1", "prenom1", new GregorianCalendar(
        1965, Calendar.JANUARY, 18).getTime());

    ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
    Validator validator = factory.getValidator();

    Set<ConstraintViolation<PersonneBean>> constraintViolations =
      validator.validate(personne);

    if (constraintViolations.size() > 0 ) {
      System.out.println("Impossible de valider les donnees du bean : ");
      for (ConstraintViolation<PersonneBean> contraintes : constraintViolations) {
        System.out.println(contraintes.getRootBeanClass().getSimpleName()+
          "." + contraintes.getPropertyPath() + " " + contraintes.getMessage());
      }
    } else {
      System.out.println("Les donnees du bean sont valides");
    }
  }
}
```

83.2.3.1. L'obtention d'un valideur

Pour obtenir une instance de Validator fournie par une implémentation de l'API, il faut utiliser une fabrique de type ValidatorFactory.

Le plus simple pour obtenir une instance de cette fabrique est d'utiliser la méthode statique buildDefaultValidatorFactory() de la classe Validation.

Il est alors possible d'utiliser la méthode getValidator() de la fabrique pour obtenir une instance de type Validator.

Exemple :

```
package com.jmdoudoux.test.validation;
```

```
import java.util.Calendar;
import java.util.GregorianCalendar;
import java.util.Set;

import javax.validation.ConstraintViolation;
import javax.validation.Validation;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;

public class TestValidation {

  public static void main(String[] args) {
...

    ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
    Validator validator = factory.getValidator();

...
  }
}
```

83.2.3.2. L'interface Validator

L'interface javax.Validation.Validator est l'élément principal de l'API de validation des contraintes.

L'interface Validator propose des méthodes pour demander la validation de données notamment :

Méthode	Rôle
Set<ConstraintViolation<T>> validate(T, Class< ?>...)	Demander la validation des données d'un bean et éventuellement de ces dépendances
Set<ConstraintViolation<T>> validateProperty(T, String, Class< ?>...)	Demander la validation de la valeur d'une propriété d'un bean. Cette méthode est utile pour la validation partielle d'un bean
Set<ConstraintViolation<T>> validateValue(T, String, Object, Class< ?>...)	Demander la validation d'une valeur par rapport à une propriété particulière d'un bean

Si la collection est vide, c'est que la validation a réussie sinon c'est que la validation a échouée et la collection contient alors la ou les raisons de l'échec sous la forme d'une occurrence pour chaque contrainte qui n'a pas été validée.

Toutes les méthodes attendent aussi un paramètre de type varargs qui peut être utilisé pour préciser les groupes à valider. Si aucun groupe n'est précisé, c'est le groupe par défaut (javax.validation.Default) qui est utilisé.

83.2.3.3. L'utilisation d'un valideur

La méthode validate() permet de demander la validation des données d'un bean et éventuellement de ces dépendances.

Exemple :

```
package com.jmdoudoux.test.validation;

import java.util.Calendar;
import java.util.GregorianCalendar;
import java.util.Set;

import javax.validation.ConstraintViolation;
import javax.validation.Validation;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;
```

```
public class TestValidationGroupe {

  public static void main(String[] args) {

    Groupe groupe = new Groupe("Mon groupe");

    PersonneBean personne = new PersonneBean(null, null, new GregorianCalendar(
        2065, Calendar.JANUARY, 18).getTime());

    groupe.ajouter(personne);

    ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
    Validator validator = factory.getValidator();

    Set<ConstraintViolation<Groupe>> constraintViolations =
      validator.validate(groupe);

    if (constraintViolations.size() > 0 ) {
      System.out.println("Impossible de valider les donnees du bean : ");
      for (ConstraintViolation<Groupe> contraintes : constraintViolations) {
        System.out.println(contraintes.getRootBeanClass().getSimpleName()+
          "." + contraintes.getPropertyPath() + " " + contraintes.getMessage());
      }
    } else {
      System.out.println("Les donnees du groupe sont valides");
    }

  }
}
```

La méthode validateProperty() permet de valider la valeur d'une propriété d'un bean.

Exemple :

```
package com.jmdoudoux.test.validation;

import java.util.Calendar;
import java.util.GregorianCalendar;
import java.util.Set;

import javax.validation.ConstraintViolation;
import javax.validation.Validation;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;

public class TestValidationProperty {

  public static void main(String[] args) {

    MonBean monBean = new MonBean(new GregorianCalendar(1980,
                                                        Calendar.DECEMBER,
                                                        25).getTime());

    ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
    Validator validator = factory.getValidator();

    Set<ConstraintViolation<MonBean>> constraintViolations =
      validator.validateProperty(monBean,
                                 "maValeur");
    validator.validate(monBean);

    if (constraintViolations.size() > 0) {
      System.out.println("Impossible de valider les donnees du bean : ");
      for (ConstraintViolation<MonBean> contraintes : constraintViolations) {
        System.out.println("   "
            + contraintes.getRootBeanClass().getSimpleName() + "."
            + contraintes.getPropertyPath() + " " + contraintes.getMessage());
      }
    } else {
      System.out.println("Les donnees du bean sont validees");
    }
```

```
  }
}
```

La méthode validateValue() permet de valider la valeur d'une propriété particulière d'un bean.

Exemple :

```
package com.jmdoudoux.test.validation;

import java.util.Calendar;
import java.util.Date;
import java.util.GregorianCalendar;
import java.util.Set;

import javax.validation.ConstraintViolation;
import javax.validation.Validation;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;

public class TestValidationValue {

  public static void main(String[] args) {

    Date valeur = new GregorianCalendar(1980, Calendar.DECEMBER, 25).getTime();

    ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
    Validator validator = factory.getValidator();

    Set<ConstraintViolation<MonBean>> constraintViolations =
      validator.validateValue(MonBean.class,
                              "maValeur",
                              valeur);
    if (constraintViolations.size() > 0) {
      System.out.println("Impossible de valider la valeur de la donnee du bean : ");
      for (ConstraintViolation<MonBean> contraintes : constraintViolations) {
        System.out.println("  "
            + contraintes.getRootBeanClass().getSimpleName() + "."
            + contraintes.getPropertyPath() + " " + contraintes.getMessage());
      }
    } else {
      System.out.println("La valeur de la donnees du bean est validee");
    }
  }
}
```

Remarque : la validation des dépendances déclarées avec l'annotation @Valid n'est effective qu'avec la méthode validate().

83.2.3.4. L'interface ConstraintViolation

L'interface ConstraintViolation<T> encapsule les informations relatives à l'échec de la validation d'une contrainte.

Elle propose plusieurs méthodes pour obtenir ces données :

Méthode	Rôle
String getMessage()	Renvoie le message d'erreur interpolé
String getMessageTemplate()	Renvoie le message d'erreur non interpolé (généralement la valeur de l'attribut message de la contrainte)
T getRootBean()	Renvoie le bean racine qui a été validé (c'est l'objet qui a été passé en paramètre de la méthode validate() de la classe Validator)
Class<T> getRootBeanClass()	Renvoie la classe du bean racine qui a été validé

Object etLeafBean()	Renvoie l'objet sur lequel la contrainte est appliquée
Object getInvalidValue()	Renvoie la valeur qui a fait échouer la contrainte (la valeur passée en paramètre de la méthode isValid() de la classe ConstraintValidator)
ConstraintDescriptor<?> getConstraintDesrcriptor()	Renvoie un objet qui encapsule la contrainte

83.2.3.5. La mise en oeuvre des groupes

Comme les contraintes sont définies au niveau des entités du domaine et que les validations peuvent se faire dans toutes les couches de l'application, il faut que les contraintes puissent s'appliquer partout.

Ce n'est pas toujours le cas : c'est possible pour des contrôles de surface mais pour des contraintes plus compliquées ce n'est pas toujours réalisable (par exemple si un accès à la base de données est nécessaire, ...)

De plus, toutes les contraintes d'un bean ne peuvent pas être validée en même temps. Par exemple, un bean qui encapsule les données d'un assistant comportant plusieurs pages. Pour valider les données de la première page avant de passer à la seconde, une validation de l'intégralité des contraintes du bean n'est pas possible puisque les données des autres pages ne sont pas encore renseignées.

Enfin, certaines contraintes ne peuvent être réalisées dans toutes les couches car elles sont trop coûteuses par exemple en ressources ou en temps de traitement.

L'API Bean Validation adresse ces problématiques au travers de la notion de groupes qui contiennent les contraintes à valider.

Les groupes (groups) permettent de restreindre l'ensemble des contraintes qui seront testées durant une validation.

Les groupes sont des types (interfaces ou classes) ce qui permet un typage fort, de faire de l'héritage, de les documenter avec Javadoc et autorise le refactoring grâce à un IDE. Ceci permet de définir une hiérarchie de groupes. Le plus simple est d'utiliser une interface de type marqueur.

Exemple :

```
package com.jmdoudoux.test.validation;

public interface AssistantEtape1 {

}

package com.jmdoudoux.test.validation;

public interface AssistantEtape2 {

}

package com.jmdoudoux.test.validation;

public interface AssistantEtape3 {

}
```

La notion de groupe permet de donner une flexibilité à la validation en proposant d'indiquer quelles contraintes doivent être vérifiées lors de la validation. Ainsi, le ou les groupes à valider sont précisés au moment de la demande de validation des contraintes du bean.

L'attribut groups de l'annotation d'une contrainte permet de faire une validation partielle du bean : elle précise le ou les groupes qui sont concernés lors d'un validation.

Exemple :

```
package com.jmdoudoux.test.validation;

import javax.validation.constraints.NotNull;

public class DonneesAssistantBean {

  /**
   *  donnees saisie à l'étape 1 de l'assitant
   */
  private String donnees1;

  /**
   *  donnees saisie à l'étape 2 de l'assitant
   */
  private String donnees2;

  /**
   *  donnees saisie à l'étape 3 de l'assitant
   */
  private String donnees3;

  @NotNull(groups={AssistantEtape1.class, AssistantEtape2.class, AssistantEtape3.class})
  public String getDonnees1() {
    return donnees1;
  }

  public void setDonnees1(String donnees1) {
    this.donnees1 = donnees1;
  }

  @NotNull(groups={AssistantEtape2.class, AssistantEtape3.class})
  public String getDonnees2() {
    return donnees2;
  }

  public void setDonnees2(String donnees2) {
    this.donnees2 = donnees2;
  }

  @NotNull(groups={AssistantEtape3.class})
  public String getDonnees3() {
    return donnees3;
  }

  public void setDonnees3(String donnees3) {
    this.donnees3 = donnees3;
  }

}
```

Les groupes qui doivent être utilisés lors de la validation sont précisés grâce au paramètre parameter de type varargs des méthodes validate(), validateProperty() et validateValue() de la classe Validator.

Exemple :

```
package com.jmdoudoux.test.validation;

import java.util.Set;

import javax.validation.ConstraintViolation;
import javax.validation.Validation;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;

public class TestValidationDonneesAssistantBean {

  public static void main(String[] args) {
    ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
    Validator validator = factory.getValidator();
    DonneesAssistantBean donnees = new DonneesAssistantBean();
```

```
    donnees.setDonnees1("valeur donnees1");
    System.out.println("Validation des données de l'étape 1");
    validerDonnees(validator, donnees, AssistantEtape1.class);

    donnees.setDonnees2("valeur donnees2");
    System.out.println("Validation des données de l'étape 2");
    validerDonnees(validator, donnees, AssistantEtape2.class);

    donnees.setDonnees3("valeur donnees3");
    System.out.println("Validation des données de l'étape 3");
    validerDonnees(validator, donnees, AssistantEtape3.class);
  }

  private static void validerDonnees(Validator validator,
                                     DonneesAssistantBean donnees,
                                     Class<?>... groupes) {
    Set<ConstraintViolation<DonneesAssistantBean>> constraintViolations;
    constraintViolations = validator.validate(donnees, groupes);

    if (constraintViolations.size() > 0) {
      System.out.println("Impossible de valider les donnees du bean : ");
      for (ConstraintViolation<DonneesAssistantBean> contrainte : constraintViolations) {
        System.out.println("  " + contrainte.getRootBeanClass().getSimpleName()
            + "." + contrainte.getPropertyPath() + " "
            + contrainte.getMessage());
      }
    } else {
      System.out.println("Les donnees du bean sont validees");
    }
  }
}
```

Chaque contrainte qui n'a pas de groupe explicite est associé au groupe par défaut (javax.validation.Default).

Il est aussi possible d'utiliser les groupes pour les évaluer un par un en conditionnant l'évaluation du suivant par le succès de l'évaluation du précédent.

83.2.3.6. Définir et utiliser un groupe implicite

Il est possible d'associer plusieurs contraintes à un groupe sans avoir à déclarer le groupe explicitement dans la déclaration de chaque contrainte.

Chaque contrainte du groupe par défaut contenue dans une interface I est automatiquement associée au groupe I.

Exemple :

```
package com.jmdoudoux.test.validation;

import java.util.Date;

import javax.validation.constraints.NotNull;
import javax.validation.constraints.Past;

public interface Tracabilite {
  @NotNull
  @Past
  Date getDateCreation();

  @NotNull
  @Past
  Date getDateModif();

  @NotNull
  Long getUtilisateur();

}
```

Exemple :

```
package com.jmdoudoux.test.validation;

import java.util.Date;

import javax.validation.constraints.NotNull;

public class Operation implements Tracabilite {
  private Date dateCreation;
  private Date dateModification;
  private Long utilisateur;
  private String designation;

  public Operation(Date dateCreation, Date dateModification, Long utilisateur,
      String designation) {
    super();
    this.dateCreation = dateCreation;
    this.dateModification = dateModification;
    this.utilisateur = utilisateur;
    this.designation = designation;
  }

  @NotNull
  public String getDesignation() {
    return this.designation;
  }

  @Override
  public Date getDateCreation() {
    return this.dateCreation;
  }

  @Override
  public Date getDateModif() {
    return this.dateModification;
  }

  @Override
  public Long getUtilisateur() {
    return utilisateur;
  }
}
```

Ceci est pratique pour permettre la validation partielle d'un bean basée sur les fonctionnalités définies dans une interface.

Exemple :

```
package com.jmdoudoux.test.validation;

import java.util.Date;
import java.util.Set;

import javax.validation.ConstraintViolation;
import javax.validation.Validation;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;

public class TestValidationOperation {

  public static void main(String[] args) {

    Operation operation = new Operation(new Date(), new Date(), 1234l, null);

    ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
    Validator validator = factory.getValidator();

    System.out.println("Validation sur le groupe par defaut");
    Set<ConstraintViolation<Operation>> constraintViolations =
      validator.validate(operation);
```

```
    if (constraintViolations.size() > 0 ) {
      System.out.println("Impossible de valider les donnees du bean : ");
      for (ConstraintViolation<Operation> contraintes : constraintViolations) {
        System.out.println(contraintes.getRootBeanClass().getSimpleName()+
          "." + contraintes.getPropertyPath() + " " + contraintes.getMessage());
      }
    } else {
      System.out.println("Les donnees du groupe sont valides");
    }

    constraintViolations = validator.validate(operation, Tracabilite.class);

    System.out.println("Validation sur le groupe Tracabilite : ");

    if (constraintViolations.size() > 0 ) {
      System.out.println("Impossible de valider les donnees du bean : ");
      for (ConstraintViolation<Operation> contraintes : constraintViolations) {
        System.out.println(contraintes.getRootBeanClass().getSimpleName()+
          "." + contraintes.getPropertyPath() + " " + contraintes.getMessage());
      }
    } else {
      System.out.println("Les donnees du groupe sont valides");
    }

  }
}
```

Résultat :

```
Validation sur
le groupe par defaut
Impossible de
valider les donnees du bean :
Operation.designation
ne peut pas être nul
Validation sur
le groupe Tracabilite :
Les donnees du groupe sont valides
```

83.2.3.7. La définition de l'ordre des validations

Par défaut, une donnée est validée sans tenir compte d'un ordre vis-à-vis des groupes auxquelles la contrainte est associée.

Il peut cependant être utile de vouloir contrôler l'ordre d'évaluation des contraintes : il peut être utile de voir évaluer certaines contraintes de façon préliminaire à d'autres.

Pour définir cet ordre particulier dans la validation des groupes, il faut définir un groupe qui va définir une séquence ordonnée d'autres groupes. La définition de cette séquence ce fait avec l'annotation @GroupSequence

Exemple :

```
package com.jmdoudoux.test.validation;

import javax.validation.GroupSequence;

@GroupSequence( { MaContrainte1.class, MaContrainte2.class, MaContrainte2.class })
public @interface MonGroupeDeSequence {
}
```

Lors de l'évaluation des groupes de la séquence, dès que la validation d'un groupe échoue, les autres groupes de la séquence ne sont pas évalués.

Il faut faire attention de ne pas créer une dépendance cyclique entre la définition d'une séquence et les groupes qui composent cette séquence aussi bien directement qu'indirectement sinon une exception de type GroupDefinitionException est levée.

L'interface d'un groupe de séquence ne devra pas avoir de super interface.

83.2.3.8. La redéfinition du groupe par défaut

L'annotation @GroupsSequence sert aussi à redéfinir le groupe par défaut d'une classe. Il suffit de l'utiliser sur une classe pour remplacer le groupe par défaut (Default.class).

Comme les séquences ne peuvent pas avoir de dépendances circulaires, il n'est pas possible d'inclure le groupe Default dans une séquence.

Par contre, comme les contraintes d'une classe sont associées automatiquement au groupe, il faut obligatoirement ajouter le groupe (la classe elle-même) dans la séquence car les contraintes contenues dans la classe doivent être incluses dans la séquence qui redéfinit le groupe par défaut. Si ce n'est pas le cas, une exception de type GroupDefinitionException est levée lors de la validation de la classe ou la recherche des contraintes qu'elle contient.

83.2.4. Les contraintes standards

Les spécifications de la JSR 303 définissent un petit ensemble de contraintes que chaque implémentation doit fournir. Celles-ci peuvent être utilisées telle quelle ou dans une composition.

Il est aussi possible pour une implémentation de fournir d'autres contraintes.

La JSR 303 propose en standard plusieurs annotations pour des actions de validation communes.

Annotation	Rôle
@Null	Vérifier que la valeur du type concerné soit null
@NotNull	Vérifier que la valeur du type concerné soit non null
@AssertTrue	Vérifier que la valeur soit true
@AssertFalse	Vérifier que la valeur soit false
@DecimalMin	Vérifier que la valeur soit supérieure ou égale à celle fournie sous la forme d'une chaîne de caractères encapsulant un BigDecimal
@DecimalMax	Vérifier que la valeur soit inférieure ou égale à celle fournie sous la forme d'une chaîne de caractères encapsulant un BigDecimal
@Digits	Vérifier qu'un nombre n'a pas plus de chiffres avant et après la virgule que ceux précisés en paramètre
@Size	Vérifier que la taille de la donnée soit comprise en les valeurs min et max fournies
@Min	Vérifier que la valeur du type soit un nombre entier dont la valeur doit être supérieure ou égale à la valeur fournie en paramètre
@Max	Vérifier que la valeur du type soit un nombre entier dont la valeur doit être inférieure ou égale à la valeur fournie en paramètre
@Pattern	Vérifier la conformité d'une chaîne de caractères avec une expression régulière
@Valid	Demander la validation des objets dépendant de l'objet à valider
@Future	Vérifier que la date doit être dans le futur (postérieure à la date courante)
@Past	Vérifier que la date doit être dans le passée (antérieure à la date courante)

Ces contraintes sont dans le package javax.validation.constraints.

Les exemples des sections suivante vont utiliser la classe ci-dessous pour valider les données du bean d'exemple.

Exemple :

```
package com.jmdoudoux.test.validation;

import java.util.Set;

import javax.validation.ConstraintViolation;
import javax.validation.Validation;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;

public class TestValidationMonBean {

  public static void main(String[] args) {

    MonBean monBean = new MonBean("test");

    ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
    Validator validator = factory.getValidator();

    Set<ConstraintViolation<MonBean>> constraintViolations =
      validator.validate(monBean);

    if (constraintViolations.size() > 0 ) {
      System.out.println("Impossible de valider les donnees du bean : ");
      for (ConstraintViolation<MonBean> contraintes : constraintViolations) {
        System.out.println("  "+contraintes.getRootBeanClass().getSimpleName()+
          "." + contraintes.getPropertyPath() + " " + contraintes.getMessage());
      }
    } else {
      System.out.println("Les donnees du bean sont validees");
    }
  }
}
```

83.2.4.1. L'annotation @Null

Cette contrainte impose que la valeur du type concernée soit null. Elle peut s'appliquer sur n'importe quel type d'objet.

Exemple :

```
package com.jmdoudoux.test.validation;

import javax.validation.constraints.Null;

public class MonBean {

  @Null
  private String maValeur;

  public MonBean(String maValeur) {
    super();
    this.maValeur = maValeur;
  }

  public String getMaValeur() {
    return maValeur;
  }

  public void setMaValeur(String maValeur) {
    this.maValeur = maValeur;
  }

}
```

83.2.4.2. L'annotation @NotNull

Cette contrainte impose que la valeur du type concernée soit null. Elle peut s'appliquer sur n'importe quel type d'objet.

Exemple :

```
package com.jmdoudoux.test.validation;

import javax.validation.constraints.NotNull;

public class MonBean {

  @NotNull
  private String maValeur;

  public MonBean(String maValeur) {
    super();
    this.maValeur = maValeur;
  }

  public String getMaValeur() {
    return maValeur;
  }

  public void setMaValeur(String maValeur) {
    this.maValeur = maValeur;
  }

}
```

83.2.4.3. L'annotation @AssertTrue

Cette contrainte impose que la valeur du type concernée soit true ou null (la donnée est valide si sa valeur est null).

Exemple :

```
package com.jmdoudoux.test.validation;

import javax.validation.constraints.AssertTrue;

public class MonBean {

  @AssertTrue
  private boolean maValeur;

  public MonBean(boolean maValeur) {
    super();
    this.maValeur = maValeur;
  }

  public boolean getMaValeur() {
    return maValeur;
  }

  public void setMaValeur(boolean maValeur) {
    this.maValeur = maValeur;
  }

}
```

Elle ne peut s'appliquer que sur un type booléen (Boolean et boolean) sinon une exception de type UnexpectedTypeException est levée à la validation ou lors de la recherche des méta données.

Exemple :

```
package com.jmdoudoux.test.validation;

import javax.validation.constraints.AssertTrue;

public class MonBean {

  @AssertTrue
  private String maValeur;

  public MonBean(String maValeur) {
    super();
    this.maValeur = maValeur;
  }

  public String getMaValeur() {
    return maValeur;
  }

  public void setMaValeur(String maValeur) {
    this.maValeur = maValeur;
  }

}
```

Résultat :

```
Exception in thread "main" javax.validation.UnexpectedTypeException: No validator could
be found for type: java.lang.String
        at org.hibernate.validator.engine.ConstraintTree.verifyResolveWasUnique(ConstraintTree.
java:236)
        at org.hibernate.validator.engine.ConstraintTree.findMatchingValidatorClass(ConstraintT
ree.java:219)
        at org.hibernate.validator.engine.ConstraintTree.getInitializedValidator(ConstraintTree
.java:167)
        at org.hibernate.validator.engine.ConstraintTree.validateConstraints(ConstraintTree.jav
a:113)
        at org.hibernate.validator.metadata.MetaConstraint.validateConstraint(MetaConstraint.ja
va:121)
        at org.hibernate.validator.engine.ValidatorImpl.validateConstraint(ValidatorImpl.java:3
34)
        at org.hibernate.validator.engine.ValidatorImpl.validateConstraintsForRedefinedDefaultG
roup(ValidatorImpl.java:278)
        at org.hibernate.validator.engine.ValidatorImpl.validateConstraintsForCurrentGroup(Vali
datorImpl.java:260)
        at org.hibernate.validator.engine.ValidatorImpl.validateInContext(ValidatorImpl.java:21
3)
        at org.hibernate.validator.engine.ValidatorImpl.validate(ValidatorImpl.java:119)
        at com.jmdoudoux.test.validation.TestValidationMonBean.main(TestValidationMonBean.java:
20)
```

83.2.4.4. L'annotation @AssertFalse

Cette contrainte impose que la valeur du type concernée soit false ou null (la donnée est valide si sa valeur est null).

Exemple :

```
package com.jmdoudoux.test.validation;

import javax.validation.constraints.AssertTrue;

public class MonBean {

```

```
  @AssertFalse
  private boolean maValeur;

  public MonBean(boolean maValeur) {
    super();
    this.maValeur = maValeur;
  }

  public boolean getMaValeur() {
    return maValeur;
  }

  public void setMaValeur(boolean maValeur) {
    this.maValeur = maValeur;
  }

}
```

Elle ne peut s'appliquer que sur un type booléen (Boolean et boolean) sinon une exception de type UnexpectedTypeException est levée à la validation ou lors de la recherche des méta données.

83.2.4.5. L'annotation @Min

Cette contrainte impose que la valeur du type soit un nombre dont la valeur doit être supérieure ou égale à la valeur fournie en paramètre sous la forme d'un entier de type long. La donnée est valide si sa valeur est null.

Elle ne peut s'appliquer que sur les types : BigDecimal, BigInteger, byte, short, int, long et leurs wrappers respectifs. Le fournisseur n'a pas l'obligation de proposer une implémentation du valideur de la contrainte pour les types double et float.

Exemple :

```
package com.jmdoudoux.test.validation;

import javax.validation.constraints.Min;

public class MonBean {

  @Min(value=10)
  private int maValeur;

  public MonBean(int maValeur) {
    super();
    this.maValeur = maValeur;
  }

  public int getMaValeur() {
    return maValeur;
  }

  public void setMaValeur(int maValeur) {
    this.maValeur = maValeur;
  }

}
```

83.2.4.6. L'annotation @Max

Cette contrainte impose que la valeur du type soit un nombre dont la valeur doit être inférieure ou égale à la valeur fournie en paramètre sous la forme d'un entier de type long. La donnée est valide si sa valeur est null.

Elle ne peut s'appliquer que sur les types : BigDecimal, BigInteger, byte, short, int, long et leurs wrappers respectifs. Le fournisseur n'a pas l'obligation de proposer une implémentation du valideur de la contrainte pour les types double et float.

Exemple :

```
package com.jmdoudoux.test.validation;

import javax.validation.constraints.Min;

public class MonBean {

  @Max(value=20)
  private int maValeur;

  public MonBean(int maValeur) {
    super();
    this.maValeur = maValeur;
  }

  public int getMaValeur() {
    return maValeur;
  }

  public void setMaValeur(int maValeur) {
    this.maValeur = maValeur;
  }

}
```

83.2.4.7. L'annotation @DecimalMin

Cette contrainte impose que la valeur du type soit un nombre dont la valeur doit être supérieure ou égale à la valeur fournie en paramètre sous la forme d'une chaîne de caractères qui puisse être transformée en BigDecimal. La donnée est valide si sa valeur est null.

Elle ne peut s'appliquer que sur les types : BigDecimal, BigInteger, String, byte, short, int, long et leurs wrappers respectifs. Les types double et float ne sont pas obligatoirement supportés à cause des problèmes d'arrondis mais une implémentation peut proposer une solution par approximation de la valeur selon des règles qui lui sont propres.

Exemple :

```
package com.jmdoudoux.test.validation;

import javax.validation.constraints.DecimalMin;

public class MonBean {

  @DecimalMin(value="10.5")
  private int maValeur;

  public MonBean(int maValeur) {
    super();
    this.maValeur = maValeur;
  }

  public int getMaValeur() {
    return maValeur;
  }

  public void setMaValeur(int maValeur) {
    this.maValeur = maValeur;
  }

}
```

Si le type de données est String alors la valeur contenue doit pouvoir être convertie en BigDecimal sinon la validation échoue.

Exemple :

```
package com.jmdoudoux.test.validation;

import java.util.Set;

import javax.validation.ConstraintViolation;
import javax.validation.Validation;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;

public class TestValidationMonBean {

  public static void main(String[] args) {

    MonBean monBean = new MonBean("test");

...

  }
}
```

Exemple :

```
package com.jmdoudoux.test.validation;

import javax.validation.constraints.DecimalMin;

public class MonBean {

  @DecimalMin(value="10.5")
  private String maValeur;

  public MonBean(String maValeur) {
    super();
    this.maValeur = maValeur;
  }

  public String getMaValeur() {
    return maValeur;
  }

  public void setMaValeur(String maValeur) {
    this.maValeur = maValeur;
  }

}
```

Résultat :

```
Impossible de valider les donnees du bean :
  MonBean.maValeur doit être plus grand que 10.5
```

83.2.4.8. L'annotation @DecimalMax

Cette contrainte impose que la valeur du type soit un nombre dont la valeur doit être supérieure ou égale à la valeur fournie en paramètre sous la forme d'une chaîne de caractères qui puisse être transformée en BigDecimal. La donnée est valide si sa valeur est null.

Elle ne peut s'appliquer que sur les types : BigDecimal, BigInteger, String, byte, short, int, long et leurs wrappers respectifs. Les types double et float ne sont pas obligatoirement supportés à cause des problèmes d'arrondis mais une implémentation peut proposer une solution par approximation de la valeur selon des règles qui lui sont propre.

Exemple :

```
package com.jmdoudoux.test.validation;

import javax.validation.constraints.DecimalMax;

public class MonBean {

  @DecimalMin(value="99.9")
  private int maValeur;

  public MonBean(int maValeur) {
    super();
    this.maValeur = maValeur;
  }

  public int getMaValeur() {
    return maValeur;
  }

  public void setMaValeur(int maValeur) {
    this.maValeur = maValeur;
  }

}
```

Si le type de données est String alors la valeur contenue doit pouvoir être convertie en BigDecimal sinon la validation échoue.

83.2.4.9. L'annotation @Size

Cette contrainte impose que la taille du type soit un nombre dont la valeur doit être comprise entre les valeurs de type int fournies aux attributs min (valeur par défaut 0) et max (valeur par défaut Integer.MAX_VALUE) incluses.

Les types supportés sont :

- String : c'est la taille de la chaîne qui est évaluée
- Tableau : c'est la taille du tableau qui est évaluée
- Collection : c'est la taille de la collection qui est évaluée
- Map : c'est la taille de la Map qui est évaluée

La donnée est valide si sa valeur est null.

Exemple :

```
package com.jmdoudoux.test.validation;

import javax.validation.constraints.Size;

public class MonBean {

  @Size(min=10, max=20)
  private String maValeur;

  public MonBean(String maValeur) {
    super();
    this.maValeur = maValeur;
  }

  public String getMaValeur() {
    return maValeur;
  }

  public void setMaValeur(String maValeur) {
    this.maValeur = maValeur;
  }

}
```

83.2.4.10. L'annotation @Digits

Cette contrainte impose que la taille du type soit un nombre dont le nombre de chiffres avant et après la virgule doit être compris entre les valeurs de type int respectivement fournies aux attributs integer et fraction.

Exemple :

```
package com.jmdoudoux.test.validation;

import javax.validation.constraints.Digits;

public class MonBean {

  @Digits(integer=5, fraction=2)
  private String maValeur;

  public MonBean(String maValeur) {
    super();
    this.maValeur = maValeur;
  }

  public String getMaValeur() {
    return maValeur;
  }

  public void setMaValeur(String maValeur) {
    this.maValeur = maValeur;
  }

}
```

83.2.4.11. L'annotation @Past

Cette contrainte impose que la date vérifiée soit dans le passée.

La date actuelle est celle de la JVM. Le calendrier utilisé est celui correspondant au TimeZone et à la Locale courante.

Les types de données utilisables avec cette annotation sont Date et Calendar.

Exemple :

```
package com.jmdoudoux.test.validation;

import java.util.Date;

import javax.validation.constraints.Past;

public class MonBean {

  @Past
  private Date maValeur;

  public MonBean(Date maValeur) {
    super();
    this.maValeur = maValeur;
  }

  public Date getMaValeur() {
    return maValeur;
  }

  public void setMaValeur(Date maValeur) {
```

```
    this.maValeur = maValeur;
  }

}
```

Exemple :

```
package com.jmdoudoux.test.validation;

import java.util.Calendar;
import java.util.GregorianCalendar;
import java.util.Set;

import javax.validation.ConstraintViolation;
import javax.validation.Validation;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;

public class TestValidationMonBean {

  public static void main(String[] args) {
    MonBean monBean = new MonBean(new GregorianCalendar(1980,
      Calendar.DECEMBER, 25).getTime());
...

  }
}
```

83.2.4.12. L'annotation @Future

Cette contrainte impose que la date vérifiée soit dans le futur.

La date actuelle est celle de la JVM. Le calendrier utilisé est celui correspondant au TimeZone et à la Locale courante.

Les types de données utilisables avec cette annotation sont Date et Calendar.

Exemple :

```
package com.jmdoudoux.test.validation;

import java.util.Date;

import javax.validation.constraints.Past;

public class MonBean {

  @Futur
  private Date maValeur;

  public MonBean(Date maValeur) {
    super();
    this.maValeur = maValeur;
  }

  public Date getMaValeur() {
    return maValeur;
  }

  public void setMaValeur(Date maValeur) {
    this.maValeur = maValeur;
  }

}
```

83.2.4.13. L'annotation @Pattern

Cette contrainte permet de valider une valeur par rapport à une expression régulière. La donnée est valide si sa valeur est null. Le format de l'expression régulière est celui utilisé par la classe java.util.regex.Pattern.

Elle ne peut s'appliquer que sur une donnée de type String.

Exemple :

```
package com.jmdoudoux.test.validation;
import java.util.Date;
import javax.validation.constraints.Pattern;
public class UtilisateurBean extends PersonneBean {

  private String
digiCode;
  public
UtilisateurBean(String nom, String prenom, Date dateNaissance, String digiCode)
{
    super(nom, prenom,
dateNaissance);
    this.digiCode =
digiCode;
  }

@Pattern(regexp="\\d\\d\\d[A-F]", message="Le digicode
doit contenir 3 chiffres et une lettre entre A et F")
  public String
getDigiCode() {
    return digiCode;
  }
  public void
setDigiCode(String digiCode) {
    this.digiCode =
digiCode;
  }

}
```

L'attribut regex permet de préciser l'expression régulière sur laquelle la donnée sera validée.

L'attribut flags est un tableau de l'énumération Flag qui précise les options à utiliser par la classe Pattern. Les valeurs de l'énumération sont : UNIX_LINES, CASE_INSENSITIVE, COMMENTS, MULTILINE, DOTALL, UNICODE_CASE et CANON_EQ

83.2.5. Le développement de contraintes personnalisées

L'API Bean Validation propose des contraintes standards mais celles-ci ne peuvent pas répondre à tous les besoins en particulier pour des contraintes spécifiques. L'API propose donc de pouvoir développer et utiliser ces propres contraintes personnalisées.

La création d'une contrainte requiert plusieurs étapes :

- Créer l'annotation pour la contrainte
- Implémenter la contrainte sous la forme d'un Validator
- Définir le message d'erreur par défaut

83.2.5.1. La création de l'annotation

Une annotation est considérée comme une contrainte de validation si elle est annotée avec l'annotation javax.validation.Constraint et si sa retention policy est RUNTIME.

L'annotation est définie comme n'importe quelle annotation en utilisant l'annotation @interface et un définissant une méthode pour chaque attribut.

L'annotation de la contrainte doit être annotée avec des méta annotations comme pour la définition de toutes annotations :

- @Target({METHOD, FIELD, ANNOTATION_TYPE }) : permet de préciser le type d'entité sur laquelle l'annotation peut être utilisée
- @Retention(RUNTIME) : permet de préciser comment sera exploitée l'annotation. Dans le cas d'une contrainte de l'API Bean Validation, il faut obligatoirement utiliser la valeur RUNTIME pour permettre à l'API d'utiliser l'introspection à l'exécution
- @Constraint(validatedBy = CheckCaseValidator.class): permet de préciser la ou les classes de type Validator qui encapsulent les traitements de validation grâce à l'attribut validatedBy
- @Documented : Permet de préciser que l'utilisation de cette annotation sera incluse dans la Javadoc

L'utilisation des 3 premières méta annotations est obligatoire selon les spécifications de l'API Java Bean Validation.

Exemple :

```
@java.lang.annotation.Documented
@ConstraintValidator(value = CarteBleueValidator.class)
@java.lang.annotation.Target(value = {java.lang.annotation.ElementType.FIELD})
@java.lang.annotation.Retention(value = java.lang.annotation.RetentionPolicy.RUNTIME)
public @interface CarteBleue
{
    String message() default "";
    String[] groups() default {};
    String bankName() default "";
}
```

Les annotations standards @Target et @Retention permettent respectivement de préciser le type sur lequel l'annotation peut s'appliquer et la portée d'application de l'annotation qui doit obligatoirement être RUNTIME pour permettre à l'API de fonctionner à l'exécution.

Une annotation relative à une contrainte doit obligatoirement être annotée avec l'annotation @Constraint. Son attribut ValidateBy permet de préciser la ou les classes de type ConstraintValidator qui lui sont associées et qui contiennent les traitements de validation à instancier.

La spécification de l'API Bean Validation impose que chaque annotation d'une contrainte définisse obligatoirement trois attributs :

- message : préciser le message d'erreur en cas de violation de la contrainte (type String)
- groups : définir le ou les groupes de validation à laquelle la contrainte appartient. La valeur par défaut doit être un tableau vide de type Class<?>
- payload : fournir des données complémentaires généralement utilisées lors de l'exploitation des violations de contraintes

L'attribut message de type String permet de créer le message qui indiquera pourquoi la validation a échouée.

Il est préférable d'utiliser un ResourceBundle pour stocker les messages. Dans ce cas, la valeur de l'attribut message doit contenir la clé entourée d'accolades. Par convention, le nom de la clé doit être composé du nom pleinement qualifié de la classe concaténé à .message

Exemple :

```
String message() default "{com.acme.constraint.MyConstraint.message}";
```

L'attribut groups de type Class< ?>[] permet de définir les groupes de contraintes qui seront utilisés lors de la validation. La valeur par défaut doit être un tableau vide : dans ce cas c'est le groupe par défaut qui est utilisé.

Exemple :

```
    Class<?>[] groups() default {};
```

Les groupes ont deux utilités principales :

- Réaliser une validation partielle en précisant quelles seront les contraintes à utiliser
- Définir l'ordre dans lesquelles les contraintes seront validées

L'attribut payload de type Class< ? extends Payload>[] permet de déclarer des types qui seront associés à la contrainte. La valeur par défaut est un tableau vide.

Exemple :

```
    Class<? extends Payload>[] payload() default {};
```

Chaque classe qui est fournie en tant que payload doit implémenter l'interface Payload. Ces données sont typiquement non portables. L'utilisation d'un type permet un typage fort de l'information. Un exemple d'utilisation de ces données peut être un niveau de gravité qui permettra à la couche présentation de préciser la sévérité de la contrainte violée, chaque gravité étant représentée dans sa propre classe.

Exemple :

```
package com.jmdoudoux.test.validation;

import javax.validation.Payload;

public class Gravite {
    public static class Info implements Payload {};
    public static class Attention implements Payload {};
    public static class Erreur implements Payload {};
}
```

Il suffit alors de préciser la ou les classes comme valeur de l'attribut payload

Exemple :

```
package com.jmdoudoux.test.validation;

import javax.validation.constraints.NotNull;

public class DonneesBean {

  private String valeur1;
  private String valeur2;

  public DonneesBean(String valeur1, String valeur2) {
    super();
    this.valeur1 = valeur1;
    this.valeur2 = valeur2;
  }

  @NotNull(message="La saisie de la valeur est obligatoire", payload=Gravite.Erreur.class)
  public String getValeur1() {
    return valeur1;
  }

  public void setValeur1(String valeur1) {
    this.valeur1 = valeur1;
  }

  @NotNull(message="La saisie de la valeur est recommandée", payload=Gravite.Info.class)
  public String getValeur2() {
    return valeur2;
  }

  public void setValeur2(String valeur2) {
```

```
    this.valeur2 = valeur2;
  }
}
```

Ces classes peuvent être retrouvées dans un objet de type ConstraintDescriptor encapsulé dans les objets de type ConstraintViolation.

Exemple :

```
package com.jmdoudoux.test.validation;

import java.util.Set;

import javax.validation.ConstraintViolation;
import javax.validation.Payload;
import javax.validation.Validation;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;

public class TestValidationDonneesBean {

  public static void main(String[] args) {
    DonneesBean donneesBean = new DonneesBean(null, null);

    ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
    Validator validator = factory.getValidator();

    Set<ConstraintViolation<DonneesBean>> constraintViolations =
      validator.validate(donneesBean);

    if (constraintViolations.size() > 0 ) {
      System.out.println("Impossible de valider les donnees du bean : ");
      for (ConstraintViolation<DonneesBean> contrainte : constraintViolations) {
        String severite = "";
        for (Class<? extends Payload> gravite : contrainte.getConstraintDescriptor()
          .getPayload()) {
          severite = gravite.getSimpleName();
          break;
        }

        System.out.println(severite + "\t "+contrainte.getRootBeanClass().getSimpleName()+
          "." + contrainte.getPropertyPath() + " " + contrainte.getMessage());
      }
    } else {
      System.out.println("Les donnees du bean sont validees");
    }
  }
}
```

Résultat :

```
Impossible de valider les donnees du bean :
Info     DonneesBean.valeur2 La saisie de la valeur est recommandée
Erreur   DonneesBean.valeur1 La saisie de la valeur est obligatoire
```

Il est possible de définir des attributs spécifiques aux besoins de la contrainte.

Le nom des attributs de l'annotation d'une contrainte possède des restrictions :

- Les noms message, groups et payload sont réservés et ne peuvent donc pas être utilisé pour des attributs personnalisés
- Les noms ne peuvent pas commencer par valid

Exemple : la définition d'une contrainte avec un attribut avec une valeur par défaut

```
package com.jmdoudoux.test.validation;
```

```
import static java.lang.annotation.ElementType.ANNOTATION_TYPE;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

import javax.validation.Constraint;
import javax.validation.Payload;

@Target( { METHOD, FIELD, ANNOTATION_TYPE })
@Retention(RUNTIME)
@Constraint(validatedBy = CasseValidator.class)
@Documented
public @interface Casse {

    String message() default "La casse de la donnée est erronée";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

    boolean majuscule() default false;
}
```

Exemple : définition d'une contrainte avec un attribut obligatoire

```
package com.jmdoudoux.test.validation;

import static java.lang.annotation.ElementType.ANNOTATION_TYPE;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

import javax.validation.Constraint;
import javax.validation.Payload;

@Target( { METHOD, FIELD, ANNOTATION_TYPE })
@Retention(RUNTIME)
@Constraint(validatedBy = CasseValidator.class)
@Documented
public @interface Casse {

    String message() default "La casse de la donnée est erronée";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

    boolean majuscule();
}
```

83.2.5.2. La création de la classe de validation

Chaque contrainte doit être associée avec au moins une classe qui encapsule la logique de validation de la valeur de la donnée associée à la contrainte. Cette association est précisée grâce à l'attribut validatedBy de l'annotation @Constraint que chaque annotation d'une contrainte utilise.

Cette classe doit implémenter l'interface ConstraintValidator qui requiert deux types paramétrés via des generics :

* Le premier type est l'interface de l'annotation qui est utilisée pour invoquer le valideur
* Le second précise le type de la valeur qui sera validée par la contrainte

Si une annotation peut être utilisée sur différents types d'éléments alors il faut créer une implémentation de type ConstraintValidator pour chacun de ces types puisque le type de la donnée à valider est fourni en tant que paramètre générique.

Cette interface définit deux méthodes :

* void initialize(A constraintAnnotation)
* boolean isValid(T value, ConstraintValidatorContext context)

La méthode initialize() est invoquée une fois que la valideur est instancié : elle permet d'initialiser le valideur. Elle reçoit en paramètre l'annotation de la contrainte ce qui permet notamment d'extraire les valeurs des attributs à utiliser pour la validation. L'implémentation doit garantir que cette méthode est invoquée avant toute utilisation de la contrainte.

La méthode isValid() contient les traitements de validation de la valeur de la donnée. Le paramètre value contient la valeur de l'objet à valider. Le paramètre context encapsule les informations sur le contexte dans lequel la validation se fait. Le code de cette méthode doit être thread-safe (elle doit obligatoirement fonctionner dans un environnement multi-thread) et ne doit pas modifier la valeur de l'objet fourni en paramètre. Elle renvoie un booléen qui précise si la validation a réussie ou non.

Si une exception est levée dans les méthodes initialize() ou isValid() alors celle-ci est propagée sous la forme d'une exception de type ValidationException.

La spécification recommande comme une bonne pratique dans le traitement de validation de considérer la valeur null comme valide. Ceci permet de ne pas faire double emploi avec la contrainte @NotNull qui doit être utilisée si la valeur est invalide si elle est null.

Exemple :

```
package com.jmdoudoux.test.validation;

import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;

public class CasseValidator implements ConstraintValidator<Casse, String> {

  private boolean majuscule;

  public void initialize(Casse constraintAnnotation) {
    this.majuscule = constraintAnnotation.majuscule();
  }

  public boolean isValid(String object,
                         ConstraintValidatorContext constraintContext) {

    if (object == null)
      return true;

    if (majuscule) {
      return object.equals(object.toUpperCase());
    } else {
      return object.equals(object.toLowerCase());
    }
  }

}
```

L'interface ConstraintValidationContext encapsule des données relatives au contexte qui peuvent être exploitées lors de la validation de la contrainte sur une valeur donnée pour générer un objet de type ConstaintViolation personnalisé au cas où la donnée est invalide. Elle définit plusieurs méthodes notamment :

Méthode	Rôle

void disableDefaultConstraintViolation()	Désactiver la génération par défaut de l'objet de type ConstraintViolation
String getDefaultConstraintMessageTemplate()	Retourner le message par défaut non interpolé
ConstraintViolationBuilder buildConstraintViolationWithTemplate(String messageTemplate)	

Une contrainte est associée à une ou plusieurs implémentations de l'interface ConstraintValidator. Une implémentation doit être fournie pour chaque type de données sur laquelle la contrainte peut être appliquée. Lors de l'évaluation d'une contrainte, la seule implémentation utilisée est celle correspondant au type de la donnée à valider.

La contrainte doit proposer une implémentation pour le type de l'entité sur laquelle elle est appliquée (classe ou interface, type de la donnée ou renvoyé par le getter). L'implémentation à utiliser est déterminée dynamiquement par le moteur de validation : une exception de type UnexeptedTypeException est levée si l'implémentation correspondante n'est pas trouvée ou si plusieurs sont trouvées.

Toutes les implémentations utilisables lors de la validation de la contrainte doivent être déclarées dans l'attribut validatedBy

Exemple :

```
@Target({ METHOD, FIELD, ANNOTATION_TYPE })
@Retention(RUNTIME)
@Constraint(validatedBy = {MaContrainteValidatorPourString.class,
                           MaContrainteValidatorPourDate.class})
@Documented
public @interface MaContrainte {

    String message() default "{com.jmdoudoux.test.validation.MaContrainte.message}";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

    String parametre();

    @Target({ METHOD, FIELD, ANNOTATION_TYPE})
    @Retention(RUNTIME)
    @Documented
    @interface List {
        MaContrainte[] value();
    }

}
```

Exemple :

```
package com.jmdoudoux.test.validation;

import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;


public class MaContrainteValidatorPourString implements
             ConstraintValidator<MaContrainte, String>  {

  @Override
  public void initialize(MaContrainte arg0) {
    // TODO A coder
  }

  @Override
  public boolean isValid(String arg0, ConstraintValidatorContext arg1) {
    // TODO A coder
    return false;
  }

```

```
}
```

Exemple :

```
package com.jmdoudoux.test.validation;

import java.util.Date;

import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;


public class MaContrainteValidatorPourDate implements
  ConstraintValidator<MaContrainte, Date>  {

  @Override
  public void initialize(MaContrainte arg0) {
  // TODO A coder
  }

  @Override
  public boolean isValid(Date arg0, ConstraintValidatorContext arg1) {
    // TODO A coder
    return false;
  }

}
```

83.2.5.3. Le message d'erreur

Il est nécessaire de définir un message d'erreur par défaut qui sera utilisé s'il y a une violation de la contrainte lors de la validation d'une valeur d'un bean.

La valeur du message peut être en dur mais il est recommandé d'utiliser un ResourceBundle pour permettre notamment d'internationaliser le message.

L'API propose un mécanisme d'interpolation pour permettre une détermination dynamique du message grâce à :

- L'utilisation d'un ResourceBundle
- Des placeholders qui seront remplacés par la valeur correspondante d'un attribut de la contrainte

Pour que l'API recherche le message dans un ResourceBundle, il faut mettre comme valeur de message la clé correspondante entourée par des accolades. Par défaut, l'API recherche les messages dans un fichier nommé ValidationMessages.properties dans le classpath.

Par défaut, le fichier de ce ResourceBundle se nomme ValidationMessages.properties et doit être placé dans un répertoire du classpath. Par convention, il est recommandé que la clé soit composée du nom pleinement qualifié de la contrainte suivi de « .message ».

Exemple :

```
com.jmdoudoux.test.validation.Casse.message=La casse de la donnée est erronée
```

Pour préciser la clé du ResourceBundle à utiliser, il suffit dans la valeur la propriété message de mettre la clé entourée par des accolades.

Exemple :

```
package com.jmdoudoux.test.validation;

import static java.lang.annotation.ElementType.ANNOTATION_TYPE;
import static java.lang.annotation.ElementType.FIELD;
```

```
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

import javax.validation.Constraint;
import javax.validation.Payload;

@Target( { METHOD, FIELD, ANNOTATION_TYPE })
@Retention(RUNTIME)
@Constraint(validatedBy = CasseValidator.class)
@Documented
public @interface Casse {

    String message() default "{com.jmdoudoux.test.validation.Casse.message}";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

    boolean majuscule() default false;
}
```

83.2.5.4. L'utilisation d'une contrainte

L'utilisation d'une contrainte personnalisée se fait come avec des annotations standards : il suffit d'utiliser l'annotation de la contrainte dans le bean sur une des entités sur laquelle elle peut s'appliquer (classe, méthode ou champ).

Exemple :

```
package com.jmdoudoux.test.validation;

public class TestBean {

  private String codePays;

  public TestBean(String codePays) {
    super();
    this.codePays = codePays;
  }

  @Casse(majuscule=true)
  public String getCodePays() {
    return codePays;
  }

  public void setCodePays(String codePays) {
    this.codePays = codePays;
  }

}
```

La validation des beans annotées se fait avec la même API avec des annotations standards qu'avec des annotations personnalisées. L'implémentation par défaut de l'API instancie les classes de type ConstraintValidators en utilisant l'introspection.

Exemple :

```
package com.jmdoudoux.test.validation;

import java.util.Set;

import javax.validation.ConstraintViolation;
```

```
import javax.validation.Validation;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;

public class TestValidationTestBean {

  public static void main(String[] args) {

    TestBean bean = new TestBean("fr");

    ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
    Validator validator = factory.getValidator();

    Set<ConstraintViolation<TestBean>> constraintViolations =
      validator.validate(bean);

    if (constraintViolations.size() > 0 ) {
      System.out.println("Impossible de valider les donnees du bean : ");
      for (ConstraintViolation<TestBean> contraintes : constraintViolations) {
        System.out.println(contraintes.getRootBeanClass().getSimpleName()
          + "." + contraintes.getPropertyPath()
          + " " + contraintes.getMessage());
      }
    } else {
      System.out.println("Les donnees du groupe sont valides");
    }

  }
}
```

Si une contrainte est appliquée sur une entité différente, une exception de type UnexpectedTypeException est levée.

Si la définition d'une contrainte est invalide, une exception de type ContraintDefinitionException est levée lors de la validation ou lors de la recherche des métadatas.

83.2.5.5. Application multiple d'une contrainte

Il peut être utile de vouloir appliquer plusieurs fois la même contrainte sur une même donnée avec des propriétés différentes. Ce n'est bien sûre pas utile sur les contraintes @Null ou @NotNull mais cela peut être utile sur la contrainte @Pattern par exemple.

L'utilisation d'une même contrainte plusieurs fois sur une même entité peut aussi être utile par exemple pour appliquer la contrainte sur différents groupes avec différentes propriétés.

C'est une recommandation de la spécification d'associer à une contrainte une annotation correspondante qui gère une version multi-version de l'annotation. L'implémentation de cette recommandation devrait se faire au travers de la définition d'un annotation interne nommée List.

Exemple :

```
package com.jmdoudoux.test.validation;

import static java.lang.annotation.ElementType.ANNOTATION_TYPE;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

import javax.validation.Constraint;
import javax.validation.Payload;

@Target({ METHOD, FIELD, ANNOTATION_TYPE })
@Retention(RUNTIME)
```

```
@Constraint(validatedBy = MaContrainteValidator.class)
@Documented
public @interface MaContrainte {

    String message() default "{com.jmdoudoux.test.validation.MaContrainte.message}";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

    String parametre();

    @Target({ METHOD, FIELD, ANNOTATION_TYPE})
    @Retention(RUNTIME)
    @Documented
    @interface List {
        MaContrainte[] value();
    }

}
```

Pour appliquer plusieurs fois la même contrainte, il faut utiliser l'annotation interne List en lui passant comme valeur d'attribut un tableau des contraintes à appliquer.

Exemple :

```
package com.jmdoudoux.test.validation;

public class TestBean {

  private String codePays;

  public TestBean(String codePays) {
    super();
    this.codePays = codePays;
  }

  @MaContrainte.List( {
    @MaContrainte(parametre="param1", message="message d'erreur concernant param1"),
    @MaContrainte(parametre="param2", message="message d'erreur concernant param2"),
    @MaContrainte(parametre="param3", message="message d'erreur concernant param3")
  })
  public String getCodePays() {
    return codePays;
  }

  public void setCodePays(String codePays) {
    this.codePays = codePays;
  }

}
```

83.2.6. Les contraintes composées

Il est fréquemment utile de pouvoir regrouper un ensemble de contraintes sous la forme d'une composition réutilisable. Par exemple, lorsqu'un même champ est utilisé dans deux beans distincts, il n'est pas souhaitable d'avoir à dupliquer toutes les contraintes sur les champs des deux beans pour des raisons évidentes de facilité de maintenance.

Il est possible de définir des contraintes composées (Compound Constraints)

La composition de contraintes permet de rassembler plusieurs contraintes pour en former une seule. La composition de contraintes peut avoir plusieurs utilités :

- Créer une version spécifique de la contrainte

* Créer une combinaison de plusieurs annotations
* Exposer sous forme d'une seule contrainte plusieurs contraintes
* Faciliter la réutilisation de contraintes en évitant ainsi la duplication

Pour créer une composition, il faut annoter la composition avec les annotations des contraintes qui vont la composer et l'annotation @Constraint.

Une composition doit aussi définir les attributs message, groups et payload et des attributs dédiés.

Exemple :

```
package com.jmdoudoux.test.validation;

import static java.lang.annotation.ElementType.ANNOTATION_TYPE;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

import javax.validation.Constraint;
import javax.validation.Payload;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Pattern;
import javax.validation.constraints.Size;

@NotNull
@Size(min = 11, max = 11,
  message="La taille du numéro de sécurité social est invalide")
@Pattern(regexp = "[12]\\d\\d[01]\\d\\d\\d\\d\\d\\d\\d",
  message="Le format du numéro de sécurité social est invalide")
@Target( { METHOD, FIELD, ANNOTATION_TYPE })
@Retention(RUNTIME)
@Constraint(validatedBy = {})
@Documented
public @interface NumeroSecuriteSocial {

  String message() default "Le numéro de sécurité social est invalide";

  Class<?>[] groups() default {};

  Class<? extends Payload>[] payload() default {};

}
```

Lorsque la contrainte sera évaluée, toutes les contraintes qui la compose seront évaluées.

Par défaut, chaque contrainte dont la validation échoue génère une erreur.

Exemple :

```
package com.jmdoudoux.test.validation;

import java.util.Calendar;
import java.util.GregorianCalendar;
import java.util.Set;

import javax.validation.ConstraintViolation;
import javax.validation.Validation;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;

public class TestValidationAssureBean {

  public static void main(String[] args) {
```

```
    AssureBean assureBean = new AssureBean("nom1",
      "prenom1",
      new GregorianCalendar(1964, Calendar.FEBRUARY, 5).getTime(),
      "3650900000");

    ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
    Validator validator = factory.getValidator();

    Set<ConstraintViolation<AssureBean>> constraintViolations =
      validator.validate(assureBean);

    if (constraintViolations.size() > 0 ) {
      System.out.println("Impossible de valider les donnees du bean : ");
      for (ConstraintViolation<AssureBean> contraintes : constraintViolations) {
        System.out.println("  "+contraintes.getRootBeanClass().getSimpleName()+
          "." + contraintes.getPropertyPath() + " " + contraintes.getMessage());
      }
    } else {
      System.out.println("Les donnees du bean sont validees");
    }
  }
}
```

Résultat :

```
Impossible de valider les donnees du
bean :

AssureBean.numSecSoc Le format du numéro de sécurité social est invalide
  AssureBean.numSecSoc La taille du numéro de sécurité social est invalide
```

Il peut être souhaité de n'avoir qu'une seul message d'erreur si au moins une contrainte n'est pas validée. L'annotation @ReportAsSingleViolation permet de préciser que si au moins une contrainte de la composition n'est pas validée alors une seule violation est reportée et celles de la composition ne sont pas remontées.

Exemple :

```
package com.jmdoudoux.test.validation;

import static java.lang.annotation.ElementType.ANNOTATION_TYPE;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

import javax.validation.Constraint;
import javax.validation.Payload;
import javax.validation.ReportAsSingleViolation;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Pattern;
import javax.validation.constraints.Size;

@NotNull
@Size(min = 11, max = 11,
  message="La taille du numéro de sécurité social est invalide")
@Pattern(regexp = "[12]\\d\\d[01]\\d\\d\\d\\d\\d\\d\\d",
  message="Le format du numéro de sécurité social est invalide")
@Target( { METHOD, FIELD, ANNOTATION_TYPE })
@Retention(RUNTIME)
@Constraint(validatedBy = {})
@Documented
@ReportAsSingleViolation
public @interface NumeroSecuriteSocial {

  String message() default "Le numéro de sécurité social est invalide";

  Class<?>[] groups() default {};
```

```
  Class<? extends Payload>[] payload() default {};

}
```

Résultat :

```
Impossible de valider les donnees du
bean :
  AssureBean.numSecSoc Le numéro de sécurité social est invalide
```

Il est possible qu'un attribut d'une composition redéfinisse un ou plusieurs attributs des annotations utilisés dans la composition : dans ce cas, il faut l'annoter avec @OverrideAttribute ou @OverrideAttribute.List pour un tableau d'attributs.

La valeur de l'attribut de la composition annotée avec @OverridesAttibute est fournie à l'attribut précisé par l'attribut name de la contrainte du type précisé par l'attribut constraint.

Les types des attributs dans la composition et dans la ou les contraintes doivent être identiques.

Une exception de type ConstraintDefinitionException est levée lors de la validation de la contrainte ou lors de la recherche de ces métadonnées si la définition n'est pas valide.

Exemple :

```
package com.jmdoudoux.test.validation;

import static java.lang.annotation.ElementType.ANNOTATION_TYPE;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

import javax.validation.Constraint;
import javax.validation.OverridesAttribute;
import javax.validation.Payload;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

@NotNull
@Size(message="La taille du numéro de sécurité social est invalide")
//@Pattern(regexp = "[12]\\d\\d[01]\\d*",
//  message="Le format du numéro de sécurité social est invalide")
@Target( { METHOD, FIELD, ANNOTATION_TYPE })
@Retention(RUNTIME)
@Constraint(validatedBy = {})
@Documented
public @interface NumeroSecuriteSocial {

  String message() default "Le numéro de sécurité social est invalide";

  Class<?>[] groups() default {};

  Class<? extends Payload>[] payload() default {};

  @OverridesAttribute.List( {
    @OverridesAttribute(constraint=Size.class, name="min"),
    @OverridesAttribute(constraint=Size.class, name="max") } )
  int taille() default 11;

}
```

83.2.7. L'interpolation des messages

Chaque contrainte doit définir un message par défaut sous la forme d'une propriété nommée message qui doit avoir une valeur par défaut et qui décrit la raison de l'échec de la validation de la contrainte.

Ce message peut être redéfini au moment de l'utilisation de la contrainte.

L'interface MessageInterpolator définit les méthodes pour transformer un message pour qu'il soit compréhensible par un utilisateur.

Une instance de MessageInterpolator se charge de faire l'interpolation du message : cette interpolation consiste à déterminer le message en effectuant une résolution des chaînes de caractères entre accolades qui font office de paramètres.

Le message est une chaîne de caractères qui peut contenir des paramètres sont entourés par des accolades. Les chaînes de caractères entre accolades dans le message peuvent avoir plusieurs significations :

- être la clé d'un message dans le ResourceBundle : le contenu entre accolades sera remplacé par la valeur correspondante à la clé
- être le nom d'un attribut de l'annotation : le contenu entre accolades sera remplacé par la valeur correspondante à l'attribut

Résultat :

```
La valeur doit être comprise entre les valeurs {min} et {max}
{com.jmdoudoux.test.validation.monmessage}
```

Comme les accolades ont une signification particulière, elles doivent être échappées avec un caractère backslash pour être utilisées sous une forme littérale dans le message. Ce caractère lui-même doit être échappé avec un double backslash pour ne pas être interprété.

Il est possible de créer sa propre implémentation de MessageInterpolator pour des besoins spécifiques et la fournir en paramètre lors de l'instanciation de la fabrique ValidatorFactory.

83.2.7.1. L'algorithme d'une interpolation par défaut

Par défaut, MessageInterpolator suit l'algorithme suivant :

- Etape 1 : les paramètres sont recherchés dans un ResourceBundle applicatif nommé ValidationMessage.properties. Si la propriété est trouvée alors elle est remplacée dans le message. Cette étape est effectuée récursivement (car un paramètre peut contenir un paramètre) jusqu'à ce que plus aucun paramètre ne soit trouvé
- Etape 2 : les paramètres sont recherchés dans un ResourceBundle fournit par l'implémentation. Si la propriété est trouvée alors elle est remplacée dans le message. Cette étape n'est pas effectuée récursivement
- Etape 3 : si l'étape 2 a effectué un remplacement alors l'étape 1 est de nouveau effectuée
- Etape 4 : les paramètres sont extraits et ceux dont la valeur correspond à un des attributs de la contrainte sont remplacés par la valeur de cet attribut

Lors des recherches dans le ResourceBundle applicatif ou fourni par l'implémentation la locale utilisée est soit la locale fournie en paramètre de la méthode interpolate() soit la locale par défaut retournée par la méthode getDefault() de la classe Locale.

83.2.7.2. Le développement d'un MessageInterpolator spécifique

Il est possible de développer et d'utiliser son propre MessageInterpolator pour par exemple prendre en compte une Locale particulière ou obtenir les valeurs des paramètres d'une ressource particulière.

Il faut créer une classe qui implémente l'interface MessageInterpolator. Cette interface définit plusieurs méthodes :

Méthode	Rôle
String interpolate(String messageTemplate, Context context)	Interpoler le message final avec la locale par défaut
String interpolate(String messageTemplate, Context context, Locale locale)	Interpoler le message final avec la locale fournie en paramètre
ConstraintDescriptor<?> getConstraintDescriptor()	Renvoyer la contrainte dont le message est interpolé
Object getValidatedValue()	Renvoyer la valeur en cours de validation

Un objet de type Contexte encapsule des informations relatives à l'interpolation.

La méthode interpolate() de l'instance de MessageInterpolator est invoquée pour chaque contrainte dont la validation échoue.

Une implémentation de MessageInterpolator devrait être thread safe.

Pour associer un MessageInterpolator spécifique à un Validator, il faut utiliser la méthode messageInterpolator() de la classe Configuration en lui passant l'instance de MessageInterpolator à utiliser. Cet objet Configuration doit ensuite être fourni pour obtenir l'instance de ValidatorFactory.

Il n'y a qu'une seule instance de MessageInterpolator pour un Validator. Il est possible de remplacer cette instance pour une instance de Validator donnée en utilisant la méthode ValidatorFactory.usingContext().messageInterpolator().

Pour obtenir le MessageInterpolator par défaut, il faut invoquer la méthode Configuration.getDefaultMessageInterpolator().

83.2.8. Bootstrapping

Le bootstrappring permet de proposer plusieurs solutions pour obtenir une instance d'une fabrique de type ValidatorFactory qui va permettre de créer une instance de type Validator. Ces mécanismes permettent de découpler l'application de l'implémentation de l'API Bean Validation fournie par le fournisseur à utiliser.

Le bootstrapping permet :

- De gérer plusieurs implémentations
- Choisir l'implémentation à utiliser
- Configurer l'implémentation utilisée
- S'intégrer dans les conteneurs notamment ceux de Java EE 6

Les mécanismes de bootstrap mettent en oeuvre plusieurs interfaces :

- Validation : c'est le point d'entrée de l'API pour utiliser une implémentation
- ValidationProvider : interface qui définit les fonctionnalités utilisables lors du bootstrap
- ValidationProviderResolver : permet de rechercher la liste des implémentations utilisables dans le contexte d'exécution
- Configuration : encapsule les données de configuration lors de la création de l'instance de type ValidatorFactory
- ValidatorFactory : fabrique qui est instanciée par le mécanisme de bootstrap. Le rôle de cette fabrique est de créer des instances de type Validator

Le fichier META-INF/validation.xml peut aussi contenir des données de configuration pour le mécanisme de bootstrap.

La façon la plus facile pour obtenir une instance de la classe Validator est d'utiliser la méthode statique buidDefaultValidatorFactory() de la classe Validation et d'utiliser la fabrique pour créer une instance du type Validator.

Il existe plusieurs autres façons pour obtenir la fabrique :

- Utilisation du mécanisme proposé par le Java Service Provider mis en oeuvre par le fournisseur de l'implémentation
- Utilisation des méthodes statiques de la classe Validation

83.2.8.1. L'utilisation du Java Service Provider

Une implémentation de l'API Bean Validation peut être découverte grâce à l'utilisation du Java Service Provider.

Pour cela le fournisseur doit fournir un fichier nommé javax.validation.spi.ValidationProvider dans le répertoire META-INF/services du package (jar, war, ...) de l'implémentation.

Ce fichier doit contenir le nom pleinement qualifié de la classe de l'implémentation de ValidationProvider proposée par le fournisseur.

83.2.8.2. L'utilisation de la classe Validation

La classe Validation est le point d'entrée pour utiliser l'API bootstraping. Elle propose plusieurs méthodes pour obtenir de façon plus ou moins directe une instance du type ValidatorFactory.

La méthode buildDefaultValidatorFactory() permet d'obtenir l'instance de type ValidatorFactory() par défaut. Elle utilise l'implémentation par défaut de la classe ValidationProviderResolver pour déterminer l'ensemble des implémentations présentes dans le classpath.

Exemple :

```
package com.jmdoudoux.test.validation;

import javax.validation.Validation;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;

public class TestBootstrapBuildDefault {

  public static void main(String[] args) {
    ValidatorFactory fabrique = Validation.buildDefaultValidatorFactory();
    Validator validator = fabrique.getValidator();

    ...
  }
}
```

Si plusieurs implémentations sont présentes dans le classpath, il n'y a aucune garantie sur celle qui sera choisie lors de l'utilisation de la méthode buildDefaultValidatorFactory() de la classe Validation.

La méthode byDefaultProvider() permet de configurer la création d'une instance de la classe ValidatorFactory personnalisée. Cette méthode renvoie une instance de l'interface GenericBootstrap.

La méthode providerResolver() de l'interface GenericBootstrap permet éventuellement de préciser l'instance de type ValidationProviderResolver fournie en paramètre qui sera utilisée pour déterminer le ValidationProvider à utiliser.

La méthode configure() de l'interface GenericBootstrap créé une instance générique de l'interface Configuration en utilisant la méthode createGenericConfiguration() du premier ValidationProvider trouvé.

L'interface Configuration propose plusieurs méthodes pour préciser une instance de différents types d'interfaces qui seront utilisées par la fabrique (MessageInterpolator, TraversableResolver et ConstraintValidatorFactory).

Exemple :

```
package com.jmdoudoux.test.validation;

import javax.validation.Configuration;
import javax.validation.Validation;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;
import javax.validation.bootstrap.GenericBootstrap;

public class TestBootstratByDefaultProvider {

  public static void main(String[] args) {

    GenericBootstrap bootstrap = Validation.byDefaultProvider();

    Configuration<?> configuration = bootstrap.configure();
    ValidatorFactory factory = configuration.buildValidatorFactory();
    Validator validator = factory.getValidator();


  }
}
```

Il est possible de fournir sa propre instance de ValidationProviderResolver.

La méthode buildDefaultValidatorFactory() est équivalente à une invocation de Validation.byDefaultProvider().configure().buildValidatorFactory().

La méthode byProvider() permet d'obtenir une instance de l'interface ProviderSpecificBootstrap pour une instance de configuration qui soit spécifique à l'implémentation précise fournie en paramètre. Sa méthode configure() permet d'obtenir une instance de l'interface Configuration typée avec l'implémentation en utilisant la méthode createSpecializedConfiguration() de l'instance de ValidationProvider.

Cette méthode est particulièrement utile pour obtenir un instance d'une implémentation particulière alors que plusieurs implémentations sont présentes dans le classpath.

Exemple : obtenir une instance de ValidatorFactory de l'implémentation de référence

Exemple :

```
package com.jmdoudoux.test.validation;

import javax.validation.Configuration;
import javax.validation.Validation;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;
import javax.validation.bootstrap.ProviderSpecificBootstrap;

import org.hibernate.validator.HibernateValidator;
import org.hibernate.validator.HibernateValidatorConfiguration;

public class TestBootstrapByProvider {

  public static void main(String[] args) {

    ProviderSpecificBootstrap<HibernateValidatorConfiguration> psb =
      Validation.byProvider(HibernateValidator.class);
    Configuration configuration = psb.configure();
    ValidatorFactory fabrique = configuration.buildValidatorFactory();
    Validator validator = fabrique.getValidator();

  }
}
```

L'instance de l'interface Validator obtenue de la fabrique doit être thread safe et peut donc être mise en cache.

83.2.8.3. Les interfaces ValidationProvider et ValidationProviderResolver

L'interface ValidationProvider définit les fonctionnalités qu'une implémentation doit fournir pour être utilisée par l'API de bootstrap.

L'interface ValidationProviderResolver définit les fonctionnalités pour rechercher les implémentations de l'API Bean Validation.

83.2.8.3.1. L'interface ValidationProviderResolver

Par défaut, les implémentations sont déterminés en utilisant le mécanisme du Java Service Provider. Chaque fournisseur doit fournir un fichier javax.validation.spi.ValidationProvider dans le sous répertoire META-INF/services du jar qui contient le nom pleinement qualifié de la classe qui implémente l'interface javax.validation.spi.ValidationProvider.

L'implémentation par défaut de l'interface ValidationProviderResolver recherche dans le classpath toutes les implémentations qui définissent un service.

L'interface ValidationProviderResolver définit une seule méthode : getValidationProviders() qui renvoie une collection de type List<ValidationProvider<?>> qui contient la liste des implémentations utilisables.

Pour des cas spécifiques (utilisation d'un classloader spécifique comme avec OSGi, impossibilité d'utiliser le Java Service Provider, ...), il est possible de développer sa propre implémentation de l'interface ValidationProviderResolver.

83.2.8.3.2. L'interface ValidationProvider

Cette interface a pour rôle de lier l'API de bootstrap et l'implémentation.

La signature de cette interface est typé avec un generic d'un type qui doit hériter de Configuration :

public interface ValidationProvider<T extends Configuration<T>>

Cette interface définit plusieurs méthodes :

Méthodes	Rôle
T createSpecializedConfiguration(BootstrapState state)	Renvoie une instance spécifique à l'implémentation
Configuration<?> createGenericConfiguration(BootstrapState state)	Renvoie une instance de Configuration générique qui n'est donc pas liée à l'implémentation
ValidatorFactory buildValidatorFactory(ConfigurationState configurationState)	Renvoie une instance initialisée du type ValidatorFactory

Une instance de cette interface permet d'identifier une implémentation particulière de l'API Bean Validation.

Une implémentation de l'API doit fournir une classe qui implémente cette interface avec un constructeur sans argument et fournir un fichier javax.validation.spi.ValidationProvider dans le répertoire META-INF/services qui doit le nom pleinement qualifié de cette classe.

83.2.8.4. L'interface MessageInterpolator

Il est possible d'avoir besoin de sa propre implémentation de l'interface MessageInterpolator.

Il faut fournir une instance de cette classe à la méthode messageInterpolator() de l'instance de Configuration utilisée pour obtenir une instance de la ValidationFactory. Ainsi, toutes les instances de Validator créées par la fabrique utiliseront le MessageInterpolator personnalisé.

Il est recommandé qu'une implémentation délègue à la fin de ses traitements une invocation du MessageInterpolator par défaut pour garantir que les règles par défaut soient prise en compte. Pour obtenir une instance du MessageInterpolator par défaut il faut utiliser la méthode Configuration.getDefaultMessageInterpolator().

83.2.8.5. L'interface TraversableResolver

L'interface TraversableResolver a pour but de restreindre l'accès à certaines propriétés lors de la validation d'un bean. Un exemple d'utilisation peut être le besoin de ne pas valider les données d'une propriété d'un bean de type entité dont le chargement est tardif (lazy loading). Au moment de la validation du bean, les données de cette propriété peuvent ne pas être chargée, rendant leur validation erronée.

Cette interface définit plusieurs méthodes :

Méthode	Rôle
boolean isReachable(Object traversableObject, Path.Node traversableProperty, Class<?> rootBeanType, Path pathToTraversableObject, ElementType elementType);	Permet de déterminer si le moteur de validation peut accéder à la valeur de la propriété
boolean isCascadable(Object traversableObject, Path.Node traversableProperty, Class<?> rootBeanType, Path pathToTraversableObject, ElementType elementType);	Permet de déterminer si le moteur de validation peut valider la propriété marquée avec l'annotation @Valid. Cette méthode n'est invoquée que si l'invocation de la méthode isReachable() pour la propriété a renvoyée true

Il est possible de définir sa propre implémentation de l'interface TrasersableResolver et de fournir cette instance en paramètre de la méthode traversableResolver() de la Configuration utilisée pour créer la fabrique de type ValidatorFactory.

Une implémentation de l'interface TraversableResolver doit être thread safe.

83.2.8.6. L'interface ConstraintValidatorFactory

Une instance d'un valideur de contraintes est créée par une fabrique de type ConstraintValidatorFactory.

L'interface ConstraintValidatorFactory ne définit qu'une seule méthode :

Méthode	Rôle
<T extends ConstraintValidator<?,?>> T getInstance(Class<T> key)	renvoie une instance de l'interface ConstraintValidator

Il est recommandé que la fabrique utilise le constructeur par défaut pour créer l'instance. Elle ne devrait pas mettre en cache les instances créées.

Si une exception est levée dans la méthode getInstance(), celle-ci est propagée sous la forme d'une exception de type ValidationException.

83.2.8.7. L'interface ValidatorFactory

Le but d'un instance de l'interface ValidatorFactory est de proposer une fabrique pour créer et initialiser des objets de type Validator. Chaque instance de type Validator est créée pour un MessageInterpolator, un TraversableResolver et un ConstraintValidatorFactory donnés.

L'interface ValidatorFactory définit plusieurs méthodes.

La méthode getValidator() renvoie l'instance créée par la fabrique : celle-ci peut être stockée dans un pool.

La méthode getMessageInterpolator() renvoie l'instance de type MessageInterpolator utilisée par la fabrique.

La méthode getTraversalResolver() renvoie l'instance de type TraversalResolver utilisée par la fabrique.

La méthode getConstraintValidatorFactory() renvoie l'instance de type ConstraintValidatorFactory utilisée par la fabrique.

La méthode unwrap() permet un accès un objet spécifique à l'implémentation qui peut encapsuler des données complémentaires. Son utilisation rend le code non portable.

La méthode usingContext() renvoie un objet de type ValidatorContext qui peut encapsuler des informations de configuration lors de la création des instances de type Validator notamment une instance de type MessageInterpolator, TraversableResolver ou ConstraintValidatorFactory qui seront utilisées à la place de celles de la fabrique.

Un objet de type ValidatorFactory est créé par un objet de type Configuration.

Une implémentation de ValidatorFacotry doit être thread safe.

83.2.8.8. L'interface Configuration

Le but d'une instance de Configuration est de définir les différentes entités utiles à une instance de ValidatorFactory et permet de créer une telle instance en ayant sélectionné l'implémentation de l'API à utiliser.

Par défaut, l'implémentation à utiliser est déterminée en utilisant :

- l'instance spécifiée par l'utilisation de la méthode byProvider() de la classe Validation
- le contenu du fichier META-INF/validation.xml
- l'instance de type ValidatorProviderResolver fournie à la Configuration ou l'instance par défaut de cette interface

La signature de l'interface est :

public interface Configuration<T extends Configuration<T>>

Cette interface propose plusieurs méthodes notamment :

Méthodes	Rôle
T messageInterpolator(MessageInterpolator interpolator)	Définir l'instance de type MessageInterpolator qui sera utilisée par la fabrique. Si cette méthode n'est pas utilisée ou la valeur null lui est passée en paramètre alors c'est l'instance par défaut de l'implémentation qui sera utilisée
T constraintValidatorFactory(ConstraintValidatorFactory constraintValidatorFactory)	Définir l'instance de type ConstraintValidatorFactory qui sera utilisée par la fabrique. Si cette méthode n'est pas utilisée ou la valeur null lui est passée en paramètre alors c'est l'instance par défaut de l'implémentation qui sera utilisée
ValidatorFactory buildValidatorFactory()	Créer une instance de type ValidatorFactory. Le

	fournisseur à utiliser est déterminé et la méthode buildValidatorFactory de son instance de type ValidationProvider est invoquée
T ignoreXmlConfiguration()	Demander de ne pas tenir compte du contenu du fichier META-INF/validation.xml
T traversableResolver(TraversableResolver resolver)	Définir l'instance de type TraversableResolver qui sera utilisée par la fabrique. Si cette méthode n'est pas utilisée ou la valeur null lui est passée en paramètre alors c'est l'instance par défaut de l'implémentation qui sera utilisée
T addProperty(String name, String value)	Définir une propriété spécifique à l'implémentation
MessageInterpolator getDefaultMessageInterpolator()	Renvoyer l'implémentation par défaut du type MessageInterpolator
ConstraintValidatorFactory getDefaultConstraintValidatorFactory()	Renvoyer l'implémentation par défaut du type ConstraintValidatorFactory

Les méthodes qui permettent de définir des données renvoient le type en paramètre du generic pour permettre de chaîner leurs invocations.

La détermination de l'implémentation à utiliser suit plusieurs règles ordonnées :

- Utilisation de l'implémentation désignée par l'invocation de la méthode byProvider() de la classe Validation
- Utilisation des informations contenues dans le fichier META-INF/validation.xml
- Utilisation de la première implémentation renvoyée par la méthode getValidationProviders() de la classe ValidationProvider

Une instance de type Configuration est créée grâce à la classe ValidationProvider.

Une instance de Configuration est utilisée grâce à la classe Validation.

83.2.8.9. Le fichier de configuration META-INF/validation.xml

L'utilisation de ce fichier est ignorée si la méthode ignoreXMLConfiguration() de l'instance de type Configuration est invoquée.

L'utilisation du fichier META-INF/validation.xml est optionnelle mais elle permet de facilement définir quelle est l'implémentation de l'API Bean Validation qui doit être utilisée et permet de la configurer.

Un seul fichier META-INF/validation.xml doit être présent dans le classpath sinon une exception de type ValidationException est levée.

Ce fichier au format XML possède un tag racine nommé validation-config qui peut avoir plusieurs tags fils.

Tag	Rôle
default-provider	Indiquer le nom pleinement qualifié de l'implémentation du type ValidationProvider du fournisseur à utiliser
message-interpolator	Indiquer le nom pleinement qualifié de l'implémentation du type MessageInterpolator à utiliser (optionnel)
traversable-resolver	Indiquer le nom pleinement qualifié de l'implémentation du type TraversableResolver à utiliser (optionnel)
constraint-validator-factory	Indiquer le nom pleinement qualifié de l'implémentation du type ConstraintValidatorFactory à utiliser (optionnel)
constraint-mapping	Indiquer le chemin d'un fichier XML de mapping (optionnel)

property	Indiquer une propriété spécifique à une implémentation sous la forme d'une paire clé/valeur (optionnel)

Exemple : demander l'utilisation de l'implémentation de référence (Hibernate Validator)

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<validation-config
   xmlns="http://jboss.org/xml/ns/javax/validation/configuration"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation=
     "http://jboss.org/xml/ns/javax/validation/configuration validation-configuration-1.0.xsd">
   <default-provider>org.hibernate.validator.HibernateValidator</default-provider>
</validation-config>
```

83.2.9. La définition de contraintes dans un fichier XML

La suite de cette section sera développée dans une version future de ce document

83.2.10. L'API de recherche des contraintes

Les spécifications de l'API Bean Validation proposent une API qui permet de rechercher les métadonnées relatives aux contraintes définies dans les beans stockées dans un repository.

Cette API est destinée à être utilisée par des outils ou pour l'intégration dans des frameworks ou des bibliothèques.

L'interface Validator propose la méthode getConstraintsForClass() qui attend en paramètre un objet de type Class<?> et renvoie un objet immuable de type BeanDescriptor qui contient une description des contraintes de la classe concernée. La méthode getContraintsForProperty() qui attend en paramètre le nom de la propriété et renvoie un objet immuable de type BeanDescriptor qui contient une description des contraintes de la propriété concernée.

Une instance de type BeanDescriptor encapsule une description des contraintes du bean et propose un accès au métadonnées de ces contraintes.

Si la définition ou la déclaration d'une contrainte contenue dans la classe est invalide alors une exception de type ValidationException ou une de ces sous classes telles que ConstraintDefinitionException, ConstraintDeclarationException ou UnexpectedTypeException est levée.

83.2.10.1. L'interface ElementDescriptor

L'interface javax.validation.metadata.ElementDescriptor encapsule la description d'un élément de la classe qui possède une ou plusieurs contraintes.

La méthode hasContraint() renvoie un booléen qui précise si l'élément (classe, champ ou getter) possède au moins une contrainte.

La méthode Set<ConstraintDescriptor<?>> getConstraintDescriptors() renvoie une collection des descriptions des contraintes associées à l'élément.

Un objet de type ConstraintDescriptor encapsule les informations relatives à une contrainte.

La méthode findConstraints() qui renvoie une instance de type ConstraintFinder permet de rechercher des contraintes selon certains critères.

83.2.10.2. L'interface ConstraintFinder

L'interface ConstraintFinder définit plusieurs méthodes qui permettent de préciser les critères de recherche :

Méthode	Rôle
ConstraintFinder unorderedAndMatchingGroup(Class<?> ...)	Filtre sur les groupes déclarés dans la contrainte
ConstraintFinder declaredOn(ElementType ...)	Filtre sur les types d'éléments sur lesquels la contrainte est appliquée (ElementType.FIELD, ElementType.METHOD, ElementType.TYPE)
ConstraintFinder lookingAt(Scope)	Filtre sur la portée de la recherche (Scope.LOCAL_ELEMENT ou Scope.HIERARCHY)

83.2.10.3. L'interface BeanDescriptor

L'interface javax.validation.meta.BeanDescriptor qui hérite de l'interface ElementDescriptor encapsule un bean qui possèdent une ou plusieurs contraintes.

Méthode	Rôle
Boolean isBeanContrained()	renvoie un booléen qui précise si le bean contient une contrainte sur elle-même, sur une de ces propriétés ou si une de ses propriétés est marquée avec l'annotation @valid. Lorsqu'elle renvoie false, le moteur de validation ignore ce bean lors de ces traitements
PropertyDescriptor getConstraintsForProperty(String propertyName)	renvoie un objet qui contient la description de la propriété dont le nom est fourni en paramètre. Renvoie null si la propriété n'existe pas ou si elle n'a pas de contrainte ou si elle n'est pas marquée avec @Valid.
Set<PropertyDescriptor> getConstrainedProperties()	renvoie une collection des descripteurs de propriétés qui possèdent au moins une contrainte ou qui sont marquées avec l'annotation @Valid.

83.2.10.4. L'interface PropertyDescriptor

L'interface javax.validation.metadata.PropertyDescriptor qui hérite de l'interface ElementDescriptor encapsule une propriété qui possède au moins une contrainte.

Méthode	Rôle
boolean isCascaded()	Renvoie true si la propriété est marquée avec l'annotation @Valid
String getPropertyName()	Renvoie le nom de la propriété

83.2.10.5. L'interface ConstraintDescriptor

L'interface javax.validation.metadata.ConstraintDescriptor<T extends Annotation> décrit une annotation d'une contrainte.

Méthode	Rôle
T getAnnotation()	Renvoie l'annotation de la contrainte
Set<Class< ?>> getGroups	Renvoie une collection des groupes définis dans l'annotation. Si aucun groupe n'est défini alors c'est le groupe par défaut Default qui est retourné
SetClass< ? extends Payload>> getPayload()	Renvoie une collection des données supplémentaires définies dans l'annotation
List<Class< ? extends ConstraintValidator<T, ?>>> getConstraintValidatorClasses()	Renvoie une collection des classes qui implémentent la logique de validation des données
Map<String, Object> getAttributes	Renvoie une collection de type Map qui contient les attributs de l'annotation : la clé contient le nom de l'attribut, la valeur contient la valeur de l'attribut
Set<ConstraintDescriptor<?>> getComposingConstraints()	Renvoie une collection des contraintes qui sont dans la composition ou un ensemble vide si la contrainte n'est pas composée
boolean isReportAsSingleViolation()	Renvoie true si la contrainte est annotée avec @ReportAsSingleViolation

83.2.10.6. Un exemple de mise en oeuvre

L'exemple de cette section va rechercher les contraintes déclarées sur une propriété d'un bean.

Exemple :

```
package com.jmdoudoux.test.validation;

import java.lang.annotation.ElementType;
import java.util.Set;

import javax.validation.Validation;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;
import javax.validation.groups.Default;
import javax.validation.metadata.ConstraintDescriptor;
import javax.validation.metadata.PropertyDescriptor;
import javax.validation.metadata.Scope;

public class TestMetaData {

  public static void main(String[] args) {

    Set<ConstraintDescriptor<?>> contraintes = null;

    ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
    Validator validator = factory.getValidator();

    // obtenir le descripteur de la propriété
    PropertyDescriptor pd = validator.getConstraintsForClass(PersonneBean.class)
                                     .getConstraintsForProperty("nom");

    // affichage de toutes les contraintes
    contraintes = pd.getConstraintDescriptors();
```

```
    System.out.println("Nombre de contraintes=" + contraintes.size());
    afficher(contraintes);

    // recherche des contraintes
    contraintes = pd.findConstraints()
                    .declaredOn(ElementType.METHOD)
                    .unorderedAndMatchingGroups(Default.class)
                    .lookingAt(Scope.LOCAL_ELEMENT)
                    .getConstraintDescriptors();

    System.out.println("Nombre de contraintes trouvees=" + contraintes.size());
    afficher(contraintes);

  }

  private static void afficher(Set<ConstraintDescriptor<?>> constraintes) {
    for (ConstraintDescriptor<?> contrainte : constraintes) {
      System.out.println("  " + contrainte.getAnnotation().toString());
    }
  }
}
```

Résultat :

```
Nombre de contraintes=2
@javax.validation.constraints.NotNull(message={javax.validation.constraints.NotNull.message},
payload=[], groups=[])
@javax.validation.constraints.Size(message={javax.validation.constraints.Size.message},
min=0, max=50, payload=[], groups=[])
Nombre de contraintes trouvees=2
@javax.validation.constraints.NotNull(message={javax.validation.constraints.NotNull.message},
payload=[], groups=[])
@javax.validation.constraints.Size(message={javax.validation.constraints.Size.message},
min=0, max=50, payload=[], groups=[])
```

83.2.11. La validation des paramètres et de la valeur de retour d'une méthode

Les spécifications proposent une extension, dont l'implémentation par un fournisseur est optionnelle, qui permet d'utiliser les mécanismes de définition, de déclaration et de validation des contraintes au niveau des paramètres d'une méthode.

Cette extension peut être exploitée par exemple dans des aspects ou dans un interceptor.

Elle peut être utilisée pour valider les paramètres en entrée ou la valeur de retour d'une méthode lorsque celle-ci est invoquée : la validation doit être appliquée autour de l'invocation de la méthode.

La spécification définit plusieurs méthodes dans l'interface Validator pour permettre la validation des paramètres.

Méthode	Rôle
<T> Set<ConstraintViolation<T>> validateParameters(Class<T> class, Method method, Object[] parameterValues, Class<?> ... groups);	Valider chacune des valeurs passé en paramètre selon les contraintes des paramètres de la méthode
<T> Set<ConstraintViolation> validateParameter(Class<T> class, Method method, Object parameterValue, int parameterIndex, Class<?>... groups);	Valider la valeur d'un paramètre selon les contraintes du paramètre dont l'index est fourni
<T> Set<ConstraintViolation> validateReturnedValue(Class<T> class, Method method, Object returnedValue, Class<?>... groups);	Valider la valeur de retour de la méthode
<T> Set<ConstraintViolation> validateParameters(Class<T> class, Constructor constructor, Object[] parameterValues, Class<?> ... groups);	Valider chacune des valeurs passé en paramètre selon les contraintes des paramètres du constructeur

<T> Set<ConstraintViolation> validateParameter(Class<T> class, Constructor constructor, Object parameterValue, int parameterIndex, Class<?>... groups);	Valider la valeur d'un paramètre selon les contraintes définies sur le paramètre dont l'index est fourni

Les contraintes appliquées sur un paramètre de la méthode ou d'un constructeur seront évaluées. Si l'annotation @Valid est utilisée sur un paramètre alors se sont les contraintes contenues dans la classe du paramètre qui seront évaluées lors de la validation.

83.2.12. L'implémentation de référence : Hibernate Validator

La version 4.0 du projet Hibernate Validator est l'implémentation de référence de la JSR 303.

Pour mettre en oeuvre Hibernate Validator, il faut :

- télécharger l'archive sur le site du projet
- décompresser l'archive dans un répertoire du système
- ajouter le fichier hibernate-validator-4.0.1.GA.jar et les fichiers *.jar du sous répertoire lib au classpath du projet

83.2.13. Les avantages et les inconvénients

Les avantages sont :

- Standardisation des fonctionnalités de validation des données d'un bean
- Déclaration des contraintes simplifiée par des annotations
- Uniformisation de la déclaration des contraintes au niveau du bean
- Validation des contraintes à la discrétion des développeurs dans n'importe quelle couche Java d'une application

Les inconvénients sont :

- Requiert un Java 5 minimum
- Ne permet des validations que sur des JavaBeans

Il existe aussi plusieurs manques dans la JSR 303 notamment :

- Le support de la Locale du client côté serveur
- Le support de la validation des paramètres d'une méthode qui est proposé sous la forme d'une extension dont le support est optionnel
- La possibilité de définir des contraintes en utilisant une expression ou un script : cette fonctionnalité peut être développée sous la forme d'une contrainte personnalisée

83.3. D'autres frameworks pour la validation des données

Il existe plusieurs autres frameworks pour la validation des données.

Framework	Description
XWork	http://www.opensymphony.com/xwork/ Ce framework est une implémentation générique du motif de conception command Il propose des fonctionnalités pour opérer des validations en utilisant des annotations
Commons-Validator	http://commons.apache.org/validator/

	Ce framework du projet Apache Commons propose un moteur et validation et des routines de validation standards
Oval	http://oval.sourceforge.net/ Ce framework open source utilise les annotations pour la déclaration de contraintes sur n'importe quel objet Java.
iScreen	http://www.i-screen.org/ Ce framework open source utilise les annotations pour la déclaration de contraintes sur n'importe quel objet Java.
agimatec-validation	http://code.google.com/p/agimatec-validation/ Ce framework open source implémente la JSR 303

84. La communauté Java

En 2009, la plate-forme Java fête son 14eme anniversaire. Une telle durée de vie lui permet d'avoir une large communauté très productive voir peut être même trop à tel point que les débutants en Java sont souvent noyés devant une telle masse d'informations et de produits.

La communauté Java est donc très riche de part le monde. Sun contribue à la vie de cette importante communauté au travers de programme comme le JCP, SDN, java.net, ...

Divers organismes open source (Apache, Eclipse, Netbeans, CodeHaus, SpringSource, Jboss, ...) enrichissent la communauté d'APIs et d'outils particulièrement utiles et sont même moteur d'inspirations sur certaines évolutions de Java.

84.1. Le JCP

Créé en 1998, le JCP (Java Community Process) est le processus chargé de définir les évolutions de Java. Le site du JCP est à l'url www.jcp.org

Chaque évolution est traitée sous la forme de propositions nommées JSR (Java Specification Request). Le contenu d'une JSR peut être très varié, allant d'une API, d'une spécification, de la définition d'une plate-forme et même les évolutions du JCP lui même. Par exemple, voici quelques JSR :

- JSR 3 : JMX
- JSR 59 : Java 1.4
- JSR 153 : EJB 2.1
- JSR 215 : la version 2.6 du JCP lui même
- JSR 221 : JDBC 4.0
- ....

Chaque JSR possède un numéro qui est un identifiant unique. Une JSR est prise en charge par plusieurs personnes :

- le leader de la spécification (specification leader)
- un groupe de travail (experts group)

Le groupe de travail est composé d'au maximum une personne de Sun Microsystems, des collaborateurs de sociétés (de toutes tailles), de membres de communautés open source (par exemple Apache, Object Web, ...) et même de personnes individuelles. La participation au JCP est payante sauf pour les personnes individuelles.

Une spécification évolue selon plusieurs états :

- initialisation
- brouillon
- early draft review
- final
- maintenance

Chaque JSR doit fournir plusieurs éléments pour être validée :

* un document de spécifications
* une implémentation de référence (RI : reference implementation) dont le code source est diffusé
* un kit de tests de compatibilité (TCK : technology compatibility kit) : permet de valider une implémentation des spécifications

Les spécifications et l'implémentation de référence sont publiques par contre la licence du TCK est définie par le groupe de travail.

Certaines JSR ont été purement et simplement abandonnées.

84.2. Les ressources proposées par Sun

Sun Microsystems propose plusieurs sites relatifs à la technologie Java :

* http://java.sun.com/ : ce site est une véritable mine d'or pour les développeurs Java
* http://www.java.com/fr/ : ce site est destiné aux utilisateurs de la plateforme Java
* de nombreux forums de discussions notamment relatifs à Java à l'url : http://forums.sun.com/index.jspa
* plusieurs newletters à l'url : http://developers.sun.com/newsletters/
* une base de bugs et d'évolutions (Request for enhancements) qui permet d'obtenir la liste, connaître leur état et voter pour déterminer les plus importants à l'url http://bugs.sun.com/bugdatabase/

Les implémentations de Java et les logiciels open source de Sun notamment OpenJDK et GlassFish sont détaillées dans une page web dédié à l'url http://www.sun.com/software/opensource/java/

84.3. Le programme Sun Developer Network

Le programme Sun Developer Network (SDN) fournit de nombreuses ressources pour les développeurs Java comme des articles, des vidéos, des outils, ... Pour bénéficier de tout le programme, il faut préalablement s'inscrire gratuitement.

Le site à l'url : http://developers.sun.com/

84.3.1. SDN share

Ce site proposé par SDN permet aux contributeurs de partager des ressources techniques comme du code sources, des tips et des articles.

Le site est à l'url http://sdnshare.sun.com/

84.4. La communauté Java.net

Ce site, proposé par Sun, permet à la communauté de trouver un espace pour des projets relatifs à la technologie Java.

Java.net est un site communautaire qui héberge des nombreux projets open source, documentations, blogs et autres ressources.

Le site de cette communauté est à l'url http://java.net

84.5. Les JUG

Les JUG (Java User Group) sont des regroupements périodiques et généralement géographiques, de passionnés de Java dans le but de partager des expériences et des sujets techniques et de promouvoir la technologie Java.

Depuis 2008, plusieurs JUG se sont créés en France et dans les pays limitrophes.

JUG de Lorraine

http://lorrainejug.blogspot.com

JUG de Paris

http://www.parisjug.org

JUG du Luxembourg

http://www.yajug.org

JUG de Belgique

http://www.bejug.org

https://java-developpez-com.dev.java.net

JUG de Bretagne

http://www.breizhjug.org

JUG de Tours

http://www.toursjug.org

JUG de Bordeaux

http://www.bordeauxjug.org

JUG de Nantes

http://www.nantesjug.org

JUG de Nice et de Sophia Antipolis

http://www.rivierajug.org

JUG de Normandie

http://www.normandyjug.org/

Lyon JUG

http://www.lyonjug.org/

Ch'ti JUG : JUG de Lille

http://chtijug.org

Poitou-Charentes JUG

http://www.poitoucharentesjug.org

JUG de Toulouse

http://www.jugtoulouse.org

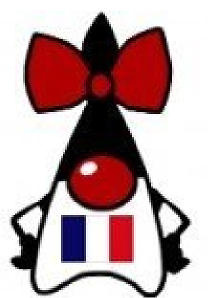
MarsJUG : JUG de Marseille

http://www.marsjug.org

AlpesJUG : JUG de Grenoble

http://www.alpesjug.fr/

JUGL : JUG Lausanne



JDuchess France

http://jduchess.org/fr/

Une liste complète des Jug mondiaux est consultable à l'url http://java.sun.com/community/usergroups/index.jsp

Le site jugevent.org recense les différents événements organisés par les Jug.

http://www.jugevents.org/jugevents/

84.6. D'autres User Groups

D'autres User Group plus spécifiques se crééent en France, notamment

- Spring User Group : http://groups.google.fr/group/sugfr
- Birt User Group : http://groups.google.fr/group/birt-user-group-france
- DDD (Domain Driven Design) User Group : http://groups.google.fr/group/dddugparis

84.7. Java BlackBelt

Ce site communautaire exploite une idée originale : il permet le passage de grade sous la forme de ceinture comme dans certains arts martiaux (en Judo par exemple) pour permettre aux développeurs Java de développer et évaluer leur connaissance, les valoriser et de les faire reconnaître.

Il propose des examens sur les trois plateformes Java mais aussi sur des frameworks et bibliothèques open source.

Certains examens sont obligatoires pour obtenir une ceinture, les autres permettent de cumuler des points.

Il est aussi possible d'obtenir des points en ajoutant des questions aux examens. Ces questions suivent un cycle d'approbation.

Il est enfin possible de créer de nouveaux examens sur des sujets non encore couverts.

Une version « corporate » permet même à des entreprises d'évaluer ces collaborateurs ou futurs collaborateurs.

Le site de Java Black Belt est à l'url : http://www.javablackbelt.com/

84.8. Les conférences

Plusieurs conférences relatives à Java ont lieu dans le monde dont quelques unes en Europe. Ces conférences sont l'occasion de rencontrer des membres de la communauté Java et d'obtenir de nombreuses informations sur les API et technologies présentes et futures relatives aux plateformes Java.

Les conférences jouent un rôle important dans la progression d'un développeur Java non seulement pour assister à des sessions thématiques techniques mais aussi rencontrer les autres membres connus ou non de la communauté Java. Lorsque l'on assiste à ses premières conférences, on y va pour assister aux sessions puis on y participe pour rencontrer d'autres amateurs de technologies Java.

84.8.1. JavaOne

JavaOne est la grande conférence annuelle organisée par Sun Microsystems au centre Moscone de San Fransisco. Cette conférence permet de découvrir de nombreuses applications et technologies relatives à Java. C'est aussi le moment pour Sun de diffuser des annonces et faire connaître des utilisations anodines de Java.

Le site de l'événement est à l'url : http://java.sun.com/javaone

84.8.2. Devoxx (ex : JavaPolis)

Devoxx (ex Javapolis) est le plus important événement indépendant européen relatif aux technologies Java : en 2008, il y a avait 3200 participants venant de 35 pays différents, 160 speakers, ... Créé en 2002, il a lieu chaque année au mois de décembre au Metropolis d'Anvers en Belgique. Il est organisé par Stephan Janssen et le Bejug. Il se déroule sur 4,5 jours et est composé de deux parties :

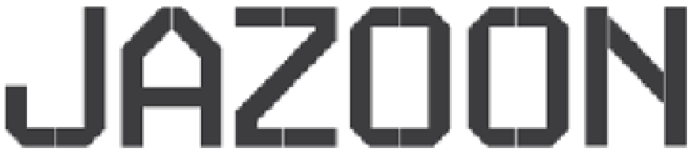
- les deux premiers jours : les universités sont des sessions longues et les tools in action
- les trois derniers jours : commencent par des keynotes puis se poursuivent par les conférences qui sont des sessions courtes

Il y a aussi un hall d'exposition, des BOFs, des quickies, ...

Le rapport qualité/prix de Devoxx est imbattable. De plus, la plupart des conférences sont, au fur et à mesure de l'année, disponibles sur le site http://www.parleys.com

Le site de l'événement est à l'url : http://www.devoxx.com

84.8.3. Jazoon



Jazoon est une conférence sur les technologies Java qui a lieu au mois de Juin.

Un des avantages de l'événement est d'être situé au milieu de l'Europe puisqu'il a lieu à Zurich en Suisse. La première session a eu lieu en 2007 et elle est reconduite chaque année.

Le site de l'événement est à l'url : http://www.jazoon.com

84.9. Webographie

http://www.developpez.com

http://www.theserverside.com/

TheServerSide est un site communautaire qui aborde les sujets relatifs aux développements d'entreprises avec Java au travers d'article et de débats souvent animés et engagés surtout ceux relatifs aux technologies de demain.

http://www.onjava.com/ est un site très riche proposé par O'Reilly

http://www.application-servers.com/ permet de suivre l'actualité du développement côté serveur

http://java.dzone.com/

http://www.dzone.com/links/ propose de nombreux liens vers de ressources sur Java mais aussi sur d'autres technologies.

http://www.javasight.com/

http://www.javarabbit.com/ : Java Certification ressource Center

http://www.javaspecialists.eu/

http://javaposse.com/

http://www.jdocs.com/ diffuse la documentation de nombreux projets open source

http://javatoolbox.com/ propose un recensement très complet des outils, frameworks, APIs pour Java

http://www.infoq.com/ est un site qui propose de nombreuses ressources sur Java mais aussi sur d'autres technologies. Les ressources Java sont directement accessibles à l'url http://www.infoq.com/java/

http://www.ibm.com/developerworks/java/ proposé par IBM, contient de nombreux articles, ressources et téléchargements.

http://www.javaworld.com/ propose de depuis très longtemps des articles techniques relatifs aux technologies Java

http://www.artima.com/index.jsp propose de nombreuses ressources sur Java mais aussi sur d'autres technologies.

http://www.jguru.com/

http://resources.corejsp.com/ propose de nombreuses ressources pour le développement web en Java

http://java.about.com/

http://java.sys-con.com/ Java Developer's Journal

http://javaboutique.internet.com/

http://www.javacrawl.com/ est un agrégateur de flux RSS sur des ressources Java

http://www.javaperformancetuning.com/

http://www.jsftutorials.net/ propose de nombreuses ressources pour JSF

84.10. Les communautés open source

La communauté open source Java est très vaste et très productive.

84.10.1. Apache - Jakarta

Le projet Jakarta de la fondation Apache regroupe un ensemble de sous projets très connus composés :

- de bibliothèques : commons, POI, Cactus, ORO, TagLibs, JCS, ...
- de frameworks : Struts, Tapestry, HiveMind, ...
- et d'outils : Tomcat, Ant, Jmeter, Maven, ...

Le site est à l'url http://jakarta.apache.org/

84.10.2. Codehaus

La fondation Codehaus propose une infrastructure pour permettre à la communauté de développer des projets open source. Parmi ces projets, il y a Xfire, izpack, mojo, sonar, m2eclipse, ...

Le site est à l'url http://codehaus.org/

84.10.3. OW2

OW2 est un consortium qui regroupe des organismes de recherche et des entreprises dans le but de développer des projets et même une plate-forme open source notamment Jonas, Joram, Enhydra, petals, easybeans, ...

Le site est à l'url http://www.ow2.org/

84.10.4. JBoss

JBoss propose une plate-forme complète incluant un serveur d'applications (jBoss AS, JBoss transaction, JBoss web services, ...), un portail (JBoss Portal), un ESB (JBoss ESB), de nombreuses bibliothèques (Hibernate, Seam, RichFaces, JGroups, RestEasy...) et des outils (Jboss Tools, ...)

Le site est à l'url http://www.jboss.org

84.10.5. Source Forge

Même s'il n'est pas dédié exclusivement à Java, SourceForge héberge de nombreux projets relatifs à Java comme le framework ZK, Dozer, FreeMarker, DBUnit, JfreeChart, Granite DS, ...

Il propose aussi d'excellents outils tels que PMD, Findbugs, SoapUI, WinMerge, MinGW, ...

Le site est à l'url http://www.sourceforge.net

Le marché des machines portables est en pleine expansion : téléphones mobiles, PDA, ... De plus en plus d'applications s'exécutent sur des machines embarquées.

Sun propose une édition particulière de Java pour ce type de développement : J2ME (Java 2 Micro Edition).

Cette partie contient les chapitres suivants :

- J2ME / Java ME : présente la plate-forme java pour le développement d'applications sur des appareils mobiles tels que des PDA ou des téléphones cellulaires
- CLDC : présente les packages et les classes de la configuration CLDC
- MIDP : propose une présentation et une mise en oeuvre du profil MIDP pour le développement d'applications mobiles
- CDC : présente les packages et les classes de la configuration CDC
- Les profils du CDC : propose une présentation et une mise en oeuvre des profils pouvant être mis en oeuvre avec la CDC
- Les autres technologies pour les applications mobiles : propose une présentation des autres technologies basées sur Java pour développer des applications mobiles

85. J2ME / Java ME

J2ME est la plate-forme Java pour développer des applications sur des appareils mobiles tels que des PDA, des téléphones cellulaires, des terminaux de points de vente, des systèmes de navigation pour voiture, ...

C'est une sorte de retour aux sources puisque Java avait été initialement développé pour piloter des appareils électroniques avant de devenir une plate-forme pour le développement et l'exécution d'applications polyvalentes.

Un environnement J2ME est composé de plusieurs éléments :

- Une machine virtuelle dédiée tenant compte des ressources limitées du matériel cible
- Un ensemble d'API de base
- Des API spécifiques

Pour répondre à la plus large gamme d'appareils cibles, J2ME est modulaire grâce à trois types d'entités qui s'utilisent par composition :

- Les configurations : définissent des caractéristiques minimales d'un large sous type de matériel, d'une machine virtuelle et d'API de base
- Les profiles : définissent des API relatives à une fonctionnalité commune à un ensemble d'appareils (exemple : interface graphique, ...)
- Les packages optionnels : définissent des API relatives à une fonctionnalité spécifique dont le support est facultatif

Ce chapitre contient plusieurs sections :

- L'historique de la plate-forme : fournit un rapide historique de la plate-forme J2ME / Java ME
- La présentation de J2ME / Java ME : présentation rapide des éléments et concepts de la plate-forme J2ME
- Les configurations : présentation des deux configurations sur lesquels la plate-forme J2ME repose
- Les profiles : présentation des profiles qui enrichissent les configurations pour un type machines ou à une fonctionnalité spécifique
- J2ME Wireless Toolkit 1.0.4 : installation et mise en oeuvre de cet outil proposé par Sun pour le développement d'applications utilisant MIDP 1.0
- J2ME wireless toolkit 2.1 : installation et mise en oeuvre de cet outil proposé par Sun pour le développement d'applications utilisant MIDP 1.0 et 2.0

85.1. L'historique de la plate-forme

Le langage Java a été développé à son origine pour la programmation d'appareils électroniques de grande consommation (langage Oak). Cependant au fil des années, Java a évolué pour être principalement utilisé pour le développement d'applications standalone et serveurs. La plate-forme Java ME peut ainsi être vue comme un retour aux buts originaux de Java.

Historiquement, Sun a proposé plusieurs plateformes pour le développement d'applications sur des machines possédant des ressources réduites, typiquement celles ne pouvant exécuter une JVM répondant aux spécifications complètes de la plate-forme Java SE.

- JavaCard (1996) : pour le développement sur des cartes à puces
- PersonnalJava (1997) : pour le développement sur des machines possédant au moins 2Mo de mémoire
- EmbeddedJava (1998) : pour des appareils avec de faibles ressources

En 1999, Sun propose de mieux structurer ces différentes plateformes sous l'appellation J2ME (Java 2 Micro Edition). Courant 2000, la plate-forme J2ME est créée pour le développement d'applications sur appareils mobiles ou embarqués tel que des téléphones mobiles, des PDA, des terminaux, ... : elle est donc le descendant des différentes plateformes antérieures relatives aux appareils mobiles. Seule le plate-forme JavaCard n'est pas incluse dans Java ME et reste à part.

Java ME cible de très nombreux appareils électroniques possédant différentes caractéristiques dans une même plate-forme.

En juin 2005, la plate-forme J2ME a été renommée, comme les autres plateformes Java, en Java ME (Java Micro Edition).

85.2. La présentation de J2ME / Java ME

La plate-forme Java Mobile Edition (J2ME/Java ME) cible le marché des appareils électroniques et embarqués tel que les pagers, les téléphones cellulaires, les PDA, les set top boxes, etc ... Elle est composée de plusieurs éléments :

- Des spécifications
- Des machines virtuelles
- Des API dédiées
- Des outils pour le développement, le déploiement et la configuration

La plate-forme Java ME cible des appareils électroniques mobiles ou embarqués dont les caractéristiques peuvent être particulièrement différentes et qui représente un nombre très important d'appareils différents. La grande difficulté est donc de définir une plate-forme qui propose des services pour le plus grand nombre d'appareils possible.

La seule solution pour répondre à cette problématique est de rendre la conception de la plate-forme modulaire. L'ensemble des appareils sur lequel peut s'exécuter une application écrite avec J2ME est tellement vaste et disparate que J2ME est composé de plusieurs parties : les configurations et les profiles qui sont spécifiés par le JCP. J2ME propose donc une architecture modulaire.

J2ME définit deux grandes familles d'appareils :

- Appareils à fonctionnalités dédiées ou limitées : ressources et interface graphique limitées, peuvent se connecter par intermittence au réseau (exemple : téléphone mobile, agenda électronique, PDA, pagers, ...)
- Appareils proposant une interface graphique riche, possédant une connexion continue au réseau (exemple : PDA haut de gamme, smartphone, set top boxes, système de navigation, ...)

La modularité de la plate-forme est assurée par trois concepts :

- Configuration : définit une spécification pour une plate-forme Java pour une des deux familles définies, une machine virtuelle et des API de base
- Profile : définit des API pour des fonctionnalités communes pour une catégorie d'appareils similaires. Un profile est définit pour une configuration sur laquelle il s'appuie et peut s'appuyer un autre profile
- Package optionnel : définit des API pour des fonctionnalités spécifiques

L'inconvénient de ce principe est qu'il déroge à la devise de Java "Write Once, Run Anywhere". Ceci reste cependant partiellement vrai pour des applications développées pour un profile particulier. Il ne faut cependant pas oublier que les types de machines cibles de J2ME sont tellement différents (du téléphone mobile au set top box), qu'il est surement impossible de trouver un dénominateur commun. Ceci associé à l'explosion du marché des machines mobiles explique les nombreuses évolutions en cours de la plate-forme.

Java ME ne définit pas un nouveau langage de programmation mais adapte la technologie Java aux appareils mobiles et embarqués.

Java ME tente de conserver autant que possible la compatibilité avec Java SE. Pour répondre aux besoins spécifiques des

appareils mobiles, Java ME remplace certaines API ou en propose de nouvelles.

Une application Java ME est organisée en plusieurs couches logicielles :

Application Java ME
Packages optionnels, API tiers
Profiles
Configuration
Machine virtuelle (VM)
Appareil

Une application est développée en reposant sur une configuration qui cible une large famille d'appareils cibles, un ou plusieurs profiles qui fournissent des fonctionnalités de base et des packages optionnels ou des API tiers pour des fonctionnalités spécifiques.

Chaque configuration peut être utilisée avec un ensemble de packages optionnels qui permet d'utiliser des technologies particulières (Bluetooth, services web, lecteur de codes barre, etc ...). Ces packages sont le plus souvent dépendant du matériel.

Les API tiers ne font pas partie de Java ME mais elles s'appuient sur elle ou l'étendent pour définir des API spécifiques à un appareil ou une fonctionnalité.

Par rapport à Java SE, Java ME utilise des machines virtuelles différentes. Certaines classes de base de l'API sont communes avec cependant de nombreuses omissions dans l'API Java ME.

Java ME définit des environnements qui servent de socles pour développer des applications portables :

- Java Technologies for Wireless Industry
- Mobile Service Architecture

Java ME est la plate-forme Java la plus récente.

De plus amples informations peuvent être obtenues sur les deux sites de Sun :

- http://wireless.java.sun.com/
- http://java.sun.com/j2me/

et sur les sites

- http://www.javamobiles.com/
- http://www.microjava.com/

85.3. Les configurations

Les configurations définissent les caractéristiques de bases d'un environnement d'exécution pour un certain type de machine possédant un ensemble de caractéristiques et de ressources similaires. Elles se composent d'une machine virtuelle et d'un ensemble d'API de base.

Deux configurations sont actuellement définies :

- CLDC (Connected Limited Device Configuration)
- CDC (Connected Device Configuration)

La CLDC 1.0 est spécifiée dans la JSR 030 : elle concerne des appareils possédant des ressources faibles (moins de 512 Kb de RAM, faible vitesse du processeur, connexion réseau limitée et intermittente) et une interface utilisateur réduite (par exemple un téléphone mobile ou un PDA "entrée de gamme"). Elle s'utilise sur une machine virtuelle KVM. La version 1.1 est le résultat des spécifications de la JSR 139 : une des améliorations les plus importantes est le support des nombres flottants.

La CDC est spécifiée dans la JSR 036 : elle concerne des appareils possédant des ressources plus importantes (au moins 2Mb de RAM, un processeur 32 bits, une meilleure connexion au réseau), par exemple un set top box ou certains PDA "haut de gamme". Elle s'utilise sur une machine virtuelle CVM.

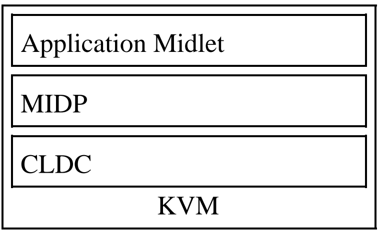
85.4. Les profiles

Les profiles se composent d'un ensemble d'API particulières à un type de machines ou à une fonctionnalité spécifique. Ils permettent l'utilisation de fonctionnalités précises et doivent être associés à une configuration. Ils permettent donc d'assurer une certaine modularité à la plate-forme J2ME.

Il existe plusieurs profiles :

Profil	Configuration	JSR	
MIDP 1.0	CLDC	37	Package javax.microedition.*
Foundation Profile	CDC	46	
Personal Profile	CDC	62	
MIPD 2.0	CLDC	118	
Personal Basis Profile	CDC	129	
RMI optional Profile	CDC	66	
Mobile Media API (MMAPI) 1.1	CLDC	135	Permet la lecture de clips audio et vidéo
PDA		75	
JDBC optional Profile	CDC	169	
Wireless Messaging API (WMA) 1.1	CLDC	120	Permet l'envoi et la réception de SMS

Les utilisations possibles des profils sont :



Application pour Palm
MIDP for Palm OS
CLDC
KVM

Application
Personal Profile
Personal Basis Profile
Foundation Profile
CDC
CVM

MIDP est un profile standard qui n'est pas défini pour une machine particulière mais pour un ensemble de machines embarquées possédant des ressources et une interface graphique limitée.

Sun a développé un profil particulier nommé KJava pour le développement spécifique sur Palm. Ce profile a été remplacé par un nouveau profil nommé MIDP for Palm OS.

Le Foundation Profile est un profil de base qui s'utilise avec CDC. Ce profil ne permet pas de développer des IHM. Il faut lui associer un des deux profils suivants :

- le Personal Basic Profile permet le développement d'application connectée avec le réseau
- le Personal Profile est un profil qui permet le développement complet d'une IHM et d'applet grâce à AWT.

PersonalJava est remplacé par le Personal Profile.

Le choix du ou des profils utilisés pour les développements est important car il conditionne l'exécution de l'application sur un type de machine supporté par le profil.

Cette multitude de profils peut engendrer un certain nombre de problème lors de l'exécution d'une application sur différents périphériques car il n'y a pas la certitude d'avoir à disposition les profils nécessaires. Pour résoudre ce problème, une spécification particulière issue des travaux de la JSR 185 et nommée Java Technology for the Wireless Industry (JTWI) a été développée. Cette spécification impose aux périphériques qui la respectent de mettre en oeuvre au minimum : CLDC 1.0, MIDP 2.0, Wireless Messaging API 1.1 et Mobile Media API 1.1. Son but est donc d'assumer une meilleure compatibilité entre les applications et les différents téléphones mobiles sur lesquelles elles s'exécutent.

85.5. J2ME Wireless Toolkit 1.0.4

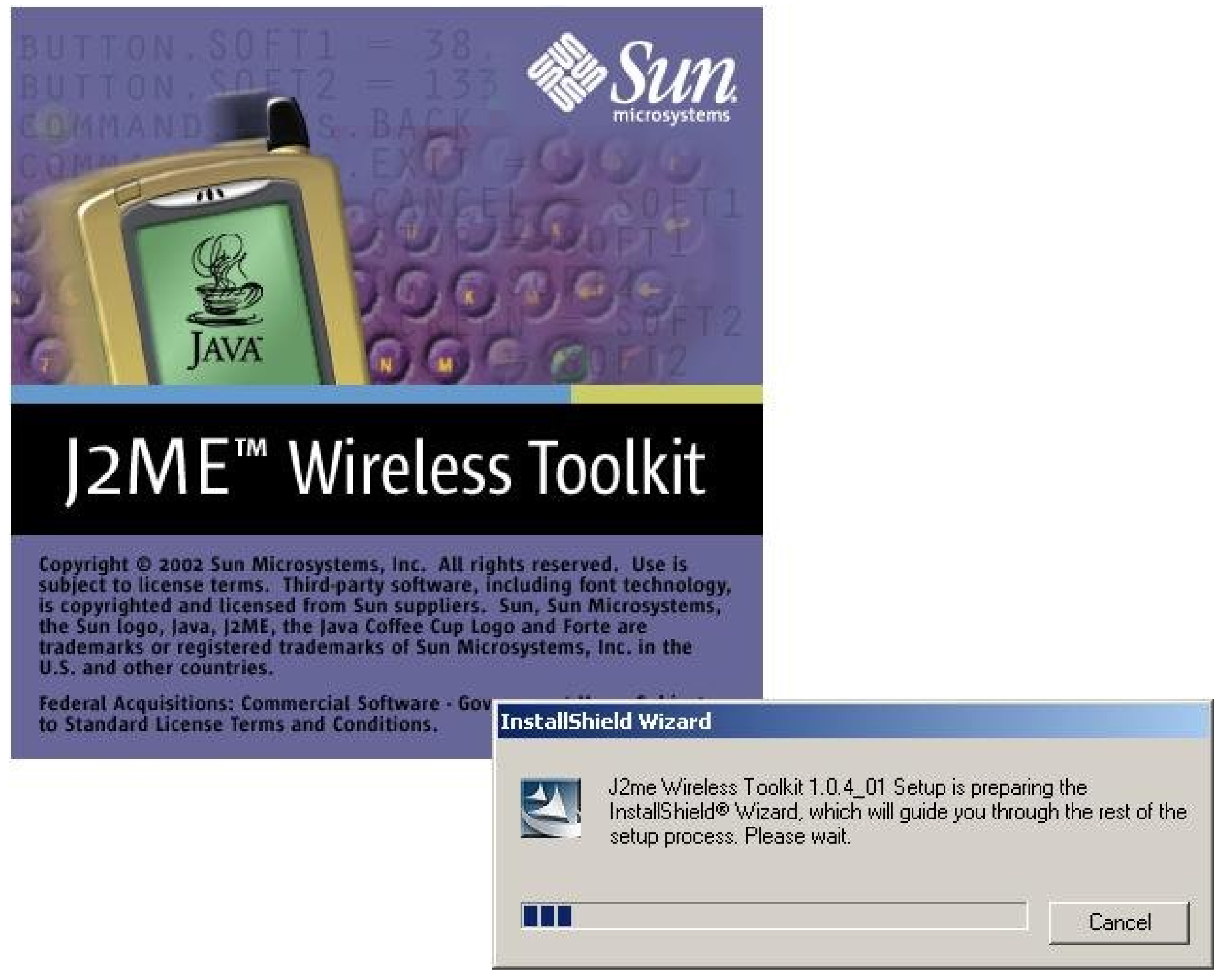
Sun propose un outil pour développer des applications J2ME utilisant CLDC/MIDP. Cet outil peut être téléchargé à l'url suivante : http://java.sun.com/products/j2mewtoolkit/index.html

La version 1.0.4 de cet outil permet de développer des applications utilisant MIDP 1.0.

85.5.1. L'installation du J2ME Wireless Toolkit 1.0.4

L'installation ci dessous concerne la version 1.0.4.

Il faut exécuter le fichier j2me_wireless_toolkit-1_0_4_01-bin-win.exe

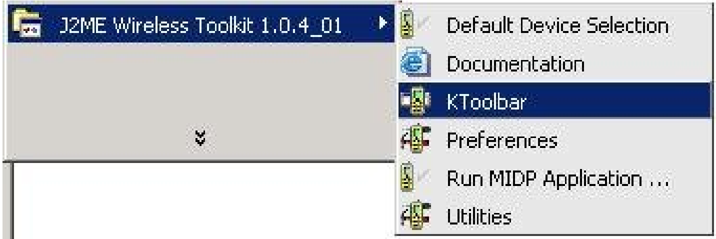


Il faut suivre les instructions suivantes, guidées par l'assistant d'installation :

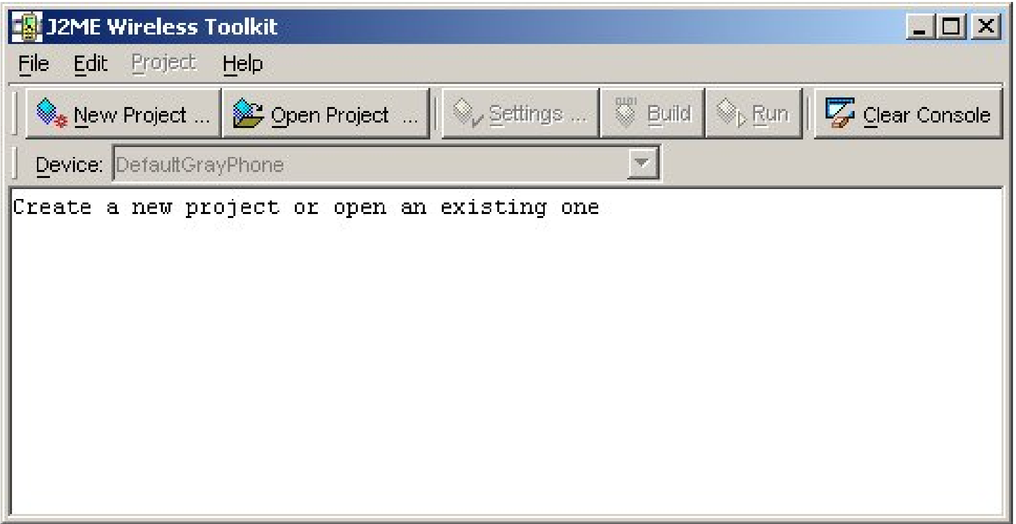
- sur la page d'accueil (welcome) , cliquez sur "Suivant"

* sur la page d'acception de la licence, lire la licence et l'approuver en cliquant sur "Yes"
* sur la page de sélection de localisation de la JVM, cliquez sur "Next" (sélectionner l'emplacement si aucune JVM n'a été détectée automatiquement)
* sélectionner l'emplacement de l'installation de l'outil et cliquez sur "Next"
* cliquez sur "Next" pour accepter le menu par défaut dans le menu "Démarrer/Programme"
* sur la page de résumé des opérations, cliquez sur "Next"
* sur le dernière page (Complete), cliquez sur "Finish"

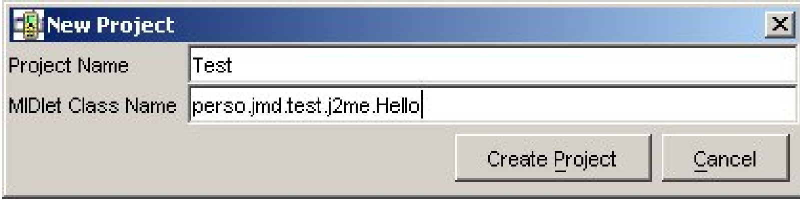
85.5.2. Les premiers pas



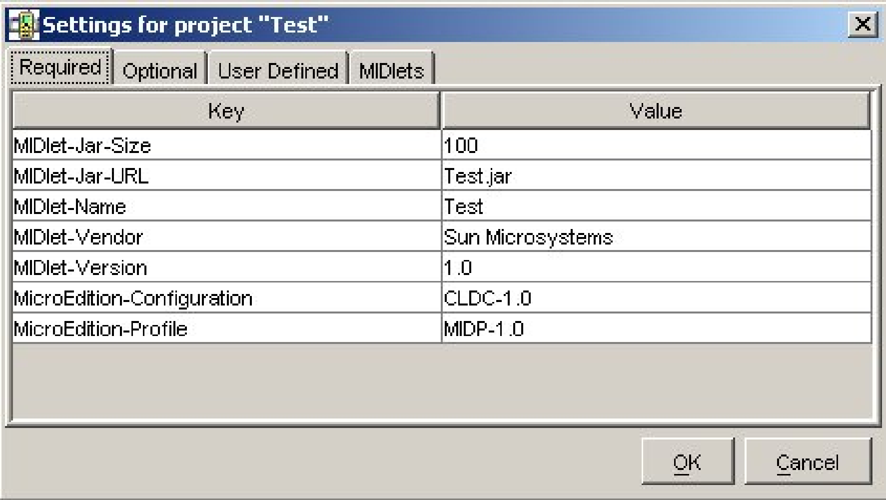
Il faut exécuter l'outil KToolBar.



Pour créer un projet, il faut cliquer sur le bouton "New Project" ou sur l'option "New Project" du menu "File".

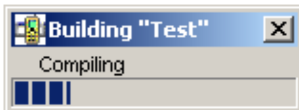


Il faut saisir le nom du projet et le nom qualifié de la midlet puis cliquer sur "Create Project".



Il faut ensuite créer la ou les classes dans le répertoire src de l'arborescence du projet.

Pour construire le projet, il faut cliquer sur le bouton "Build".



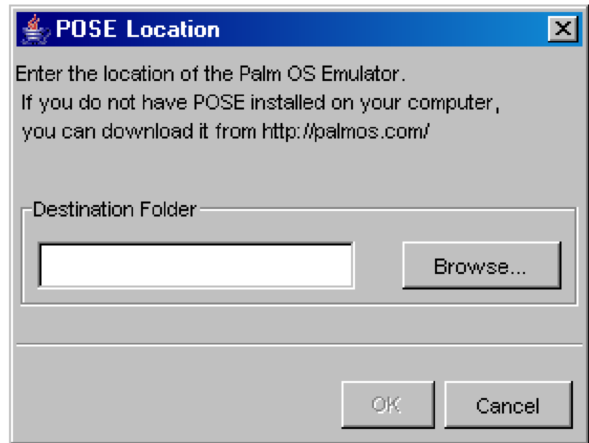
Pour exécuter le projet, il suffit de choisir le type d'émulateur à utiliser et cliquer sur le bouton "Run".

Exemple : avec l'émulateur de téléphone par défaut.

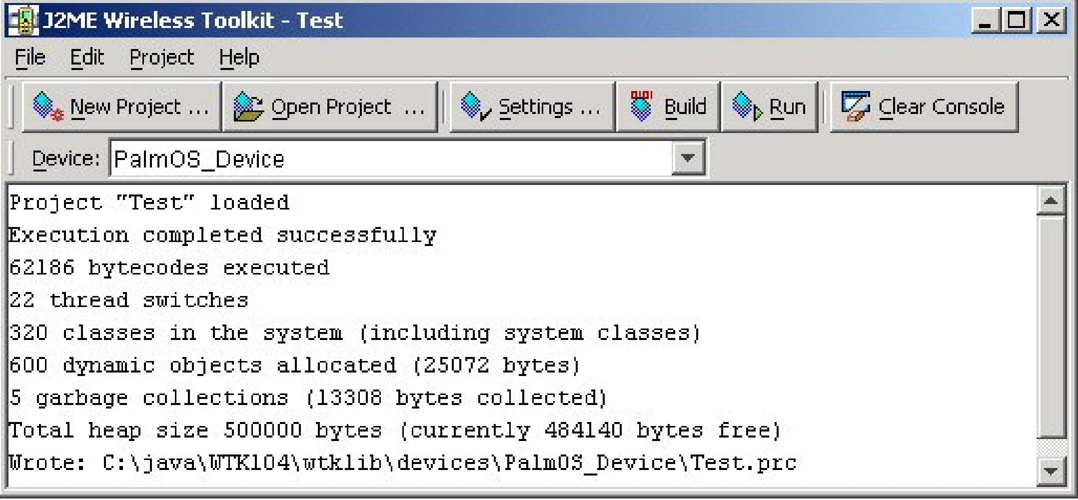
Cliquer sur l'application "Test"

puis cliquer sur le bouton entre les flèches

Il est aussi possible d'utiliser l'émulateur Palm POSE (Palm O.S. Emulator). L'outil demande le chemin d'accès à POSE,



Puis l'outil génère un fichier .prc.



Enfin, il lance l'émulateur et installe le fichier pour l'exécuter.

Pour plus de détails, voir la section sur MIDP for Palm OS.

85.6. J2ME wireless toolkit 2.1

La version 2.0 permet d'utiliser MIDP 1.0 ou 2.0 ainsi que les API optionnels Mobile Media et Wireless Messaging . Il peut être intégré dans d'autres IDE tel que Sun Studio Mobile Edition ou JBuilder.

La version 2.1 permet d'utiliser CLDC 1.1 et l'API J2ME Web service et de développer des applications pour des périphériques qui respectent les spécifications JTWI.

85.6.1. L'installation du J2ME Wireless Toolkit 2.1

La version 2.1 du J2ME Wireless Toolkit nécessite la présence sur le système d'un J2SE 1.4 minimum.

Elle permet le développement d'applications répondant aux spécifications de la JSR-185 (Java Technology for the Wireless Industry) qui inclue : CLDC 1.1, MIDP 2.0, WMA 1.1 MMAPI 1.1.

Elle permet aussi l'utilisation de la JSR-172 (J2ME Web Services Specification).

Lancer l'application j2me_wireless_toolkit-2_1-windows.exe. Un assistant guide l'utilisateur dans les différentes étapes de l'installation :

- sur la page d'accueil, cliquez sur le bouton « Next »
- sur la page « License Agreement » : lire la licence et si vous l'acceptez, cliquez sur le bouton « Yes »
- sur la page « Java Virtual Machine Location » : le programme détecte automatiquement la présence d'un JDK 1.4 ou supérieure, cliquez sur le bouton « Next »
- sur la page « Choose Destination Location » : sélectionnez le répertoire d'installation de l'application et cliquez sur le bouton « Next »
- sur la page « Select Program Folder » : saisissez ou sélectionner le dossier du menu démarrer qui va contenir les raccourcis vers l'application si celui par défaut ne convient pas. Cliquez sur le bouton « Next »
- sur la page « Start Copying files » : un résumé des options d'installation est affiché. Cliquez sur le bouton « Next »
- les fichiers de l'application sont copiés. Une fois celle ci terminée, la page « Installshield Wizard Complete » s'affiche. Cliquez sur le bouton « Finish ».

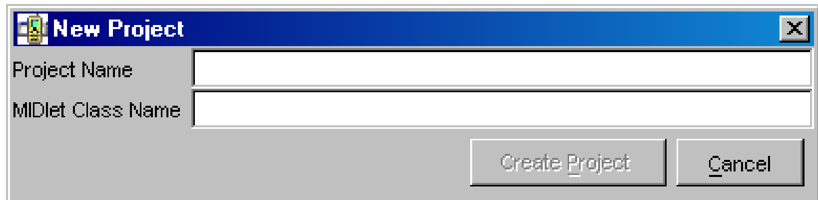
L'installation créé les répertoires suivants :

appdb\	contient les bases de données de type RMS des applications
apps\	contient les applications développées comme applications de démo
bin\	contient les outils du WTK
docs\	contient la documentation du WTK et des API
lib\	contient les bibliothèques des API

85.6.2. Les premiers pas

L'outil Ktoolbar est un petit IDE qui permet de compiler, pré-vérifier, packager et exécuter des applications utilisant le profile MIDP. Il ne permet pas l'édition du code des applications : il faut utiliser un éditeur externe pour réaliser cette tâche.

La première chose à faire pour créer une application est de créer un nouveau projet. Pour cela, il faut sélectionner l'option « File/New Project » du menu ou cliquer sur le bouton « New Project » dans la barre d'outils.

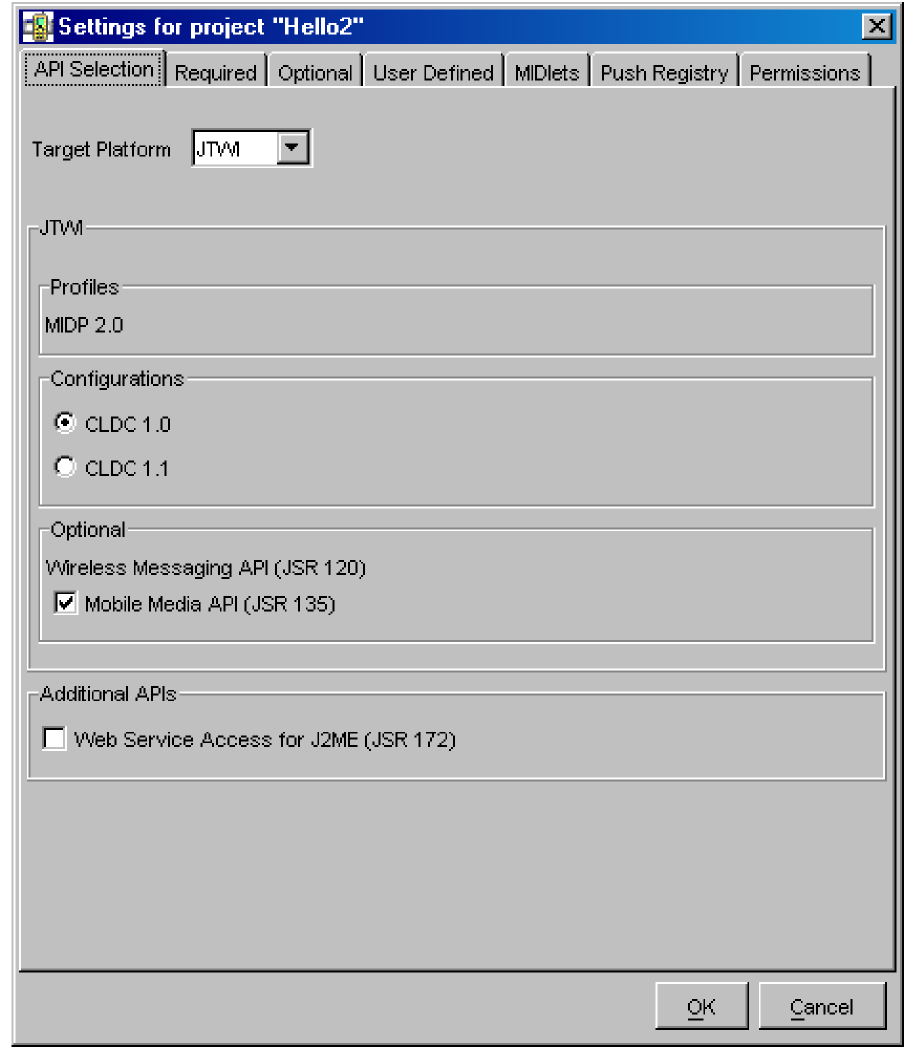


Il faut saisir le nom du projet et le nom de la classe de la Midlet.

La création du projet permet la création d'une structure de répertoires dans le sous répertoire apps du répertoire du WTK. Dans ce répertoire apps, un répertoire est créé nommé du nom du projet. Ce répertoire contient lui même plusieurs sous répertoires :

%WTK%/apps/nom_projet/bin	contient le fichier jar, jad et le fichier manifest
%WTK%/apps/nom_projet/classes	contient les classes compilées
%WTK%/apps/nom_projet/lib	contient les bibliothèques utiles à l'application
%WTK%/apps/nom_projet/res	contient les ressources utiles à l'application
%WTK%/apps/nom_projet/src	contient les sources des classes

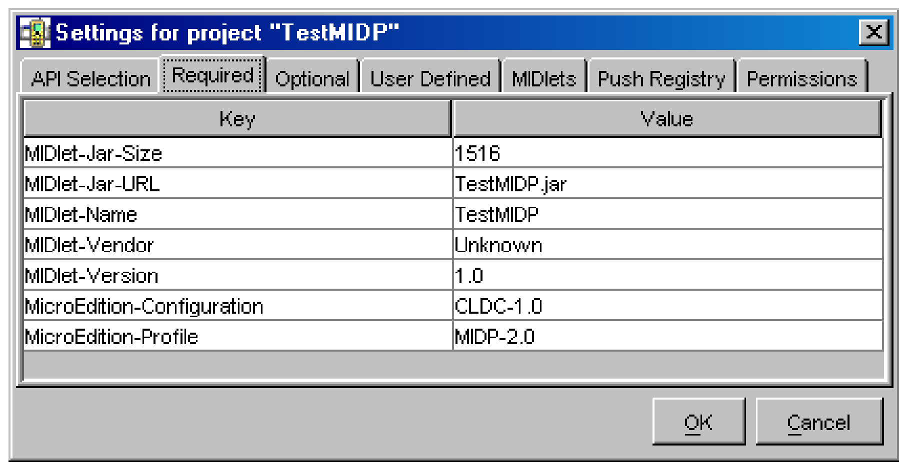
La page des propriétés du projet est différente de celle proposée dans la version 1.0. Pour l'utiliser, il faut utiliser l'option « Project/settings » ou cliquer sur le bouton « Settings » de la barre d'outils.



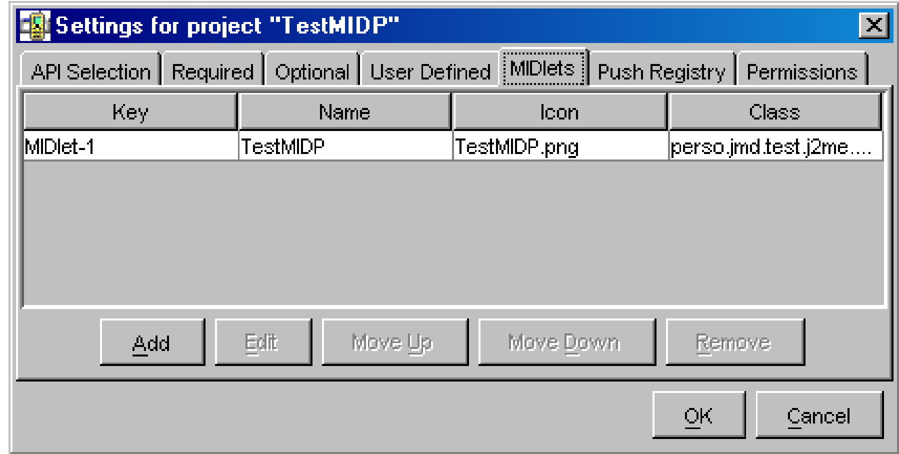
L'onglet « API sélection » permet de sélectionner la plateforme cible ainsi que les API particulières qui vont être utilisées par l'application.

Le « target plateform » permet de sélectionner le type de plate-forme cible utilisée :

- JTWI : plate-forme répondant aux spécifications de la JSR-185
- MIDP 1.0 : plate-forme composée de CLDL 1.0 et MIDP 1.0
- Custom : plate-forme personnalisée pour laquelle il faut préciser toutes les API utilisées



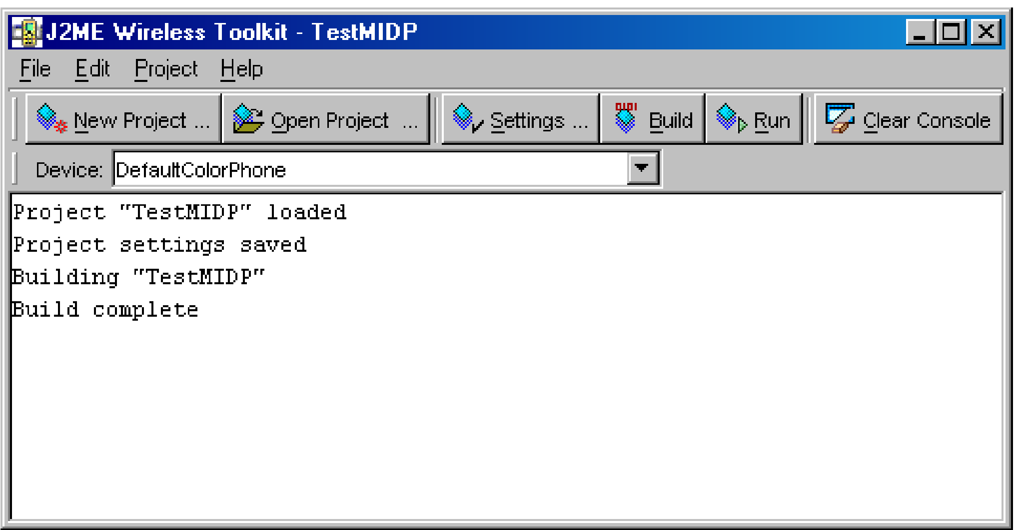
L'onglet « Required », « Optionnal » et « User defined » permet de préciser les attributs respectivement obligatoires, optionnels et particuliers à l'application dans le fichier manifest sous la forme de paire clé/valeur.



L'onglet « Midlets » permet de saisir les Midlets qui composent la suite de Midlets de l'application.

Pour créer et éditer le code des classes qui composent l'application, il faut utiliser un outil externe dans le répertoire %WTK%/apps/nom_projet/src, en respectant la structure des répertoires correspondant aux packages des classes.

La compilation et la pré-vérification des sources se fait en utilisant l'option « Build » du menu « Project » ou en cliquant sur le bouton « Build ».

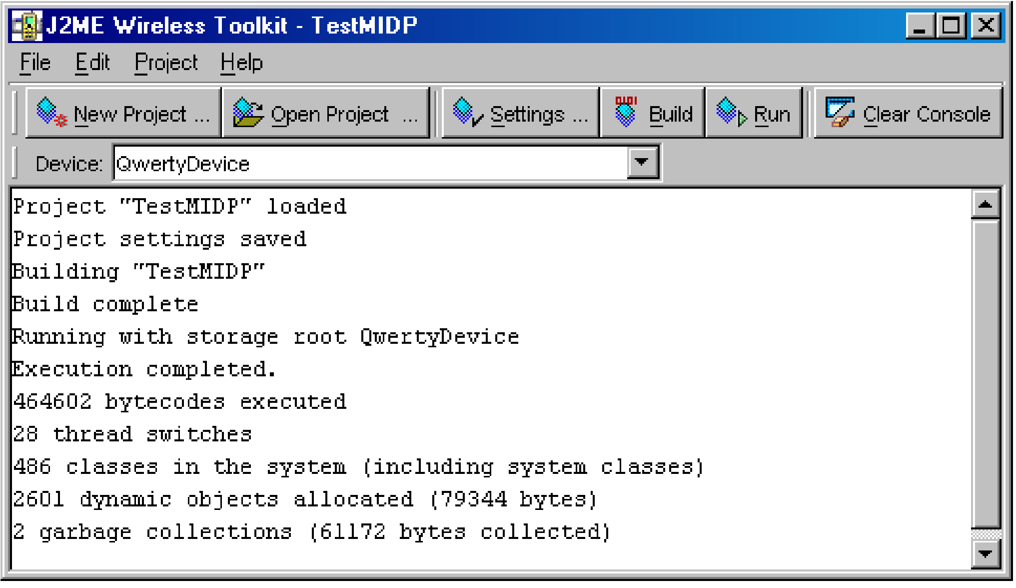


Si aucune erreur de compilation n'est détectée, il est possible d'exécuter le code en utilisant l'option « Run » du menu « Project » ou en cliquant sur le bouton « Run » de la barre d'outils.

Avant de lancer l'exécution, il est possible de sélectionner l'émulateur de périphérique (device) utilisé pour exécuter le code. Le J2ME Wireless Toolkit 2.1 est fourni avec quatre émulateurs :

- DefaultColorPhone : un téléphone mobile avec un écran couleur. C'est l'émulateur par défaut.
- DefaultGrayPhone : un téléphone mobile avec un écran monochrome
- MediaControlSkin : un téléphone mobile avec des capacités multimédia accrues (video et audio)
- QwertyDevice : un périphérique avec un clavier Qwerty

L'option « Clean » du menu « Project » permet de faire du ménage dans les fichiers temporaires générés lors des différents traitements.

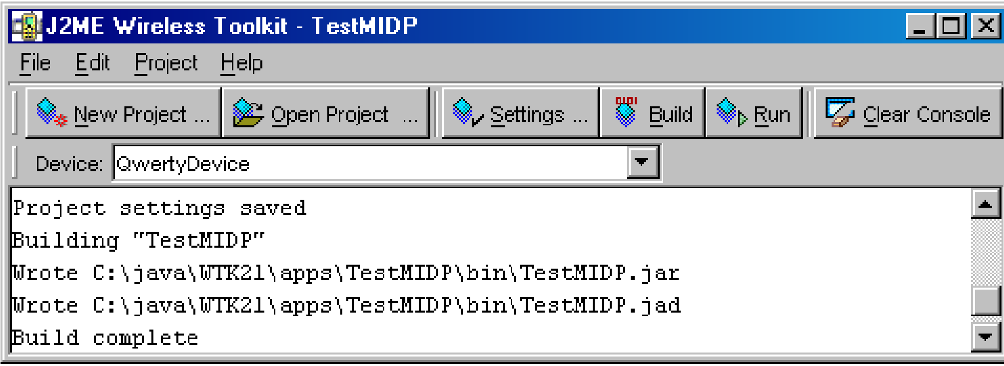


Le bouton « Clear console » permet d'effacer le contenu de la console.

L'option « Package » du menu « Project » propose deux options pour packager l'application une fois celle ci mise au point :

- « Create package » : permet de créer un package sous la forme d'un fichier .jar et .jad
- « Create obfuscated package » : permet de créer un package sous la forme d'un fichier .jad et d'un fichier .jar

plus compact et grâce à un outil tiers non fourni réalisant l'opération d'obscurcissement



86. CLDC

La suite de ce chapitre sera développée dans une version future de ce document

L'API du CLDC se compose de quatre packages :

- java.io : classes pour la gestion des entrées / sorties par flux
- java.lang : classes de base du langage java
- java.util : classes utilitaires notamment pour gérer les collections, la date et l'heure, ...
- javax.microedition.io : classes pour gérer des connections génériques

Ils ont des fonctionnalités semblables à ceux proposés par J2SE avec quelques restrictions, notamment il n'y a pas de gestion des nombres flottants dans CLDC 1.0.

De nombreuses classes sont définies dans J2SE et J2ME mais souvent elles possèdent moins de fonctionnalités dans l'édition mobile.

La version courant de CLDC est la 1.1 dont les spécifications sont les résultats des travaux de la JSR 139.

Ce chapitre contient plusieurs sections :

- Le package java.lang
- Le package java.io
- Le package java.util
- Le package javax.microedition.io

86.1. Le package java.lang

Il définit l'interface Runnable.

Il définit les classes suivantes :

Nom	Rôle
Boolean	Classe qui encapsule une valeur du type booléen
Byte	Classe qui encapsule une valeur du type byte
Character	Classe qui encapsule une valeur du type char

Class	Classe qui encapsule une classe ou une interface
Integer	Classe qui encapsule une valeur du type int
Long	Classe qui encapsule une valeur du type long
Math	Classe qui contient des méthodes statiques pour les calculs mathématiques
Object	Classe mère de toutes les classes
Runtime	Classe qui permet des interactions avec le système d'exploitation
Short	Classe qui encapsule une valeur du type short
String	Classe qui encapsule une chaîne de caractères immuable
StringBuffer	Classe qui encapsule une chaîne de caractère
System	
Thread	Classe qui encapsule un traitement exécuté dans un thread
Throwable	Classe mère de toutes les exceptions et les erreurs

Il définit les exceptions suivantes : ArithmeticException, ArrayIndexOutOfBoundsException, ArrayStoreException, ClassCastException, ClassNotFoundException, Exception, IllegalAccessException, IllegalArgumentException, IllegalMonitorStateException, IllegalThreadStateException, IndexOutOfBoundsException, InstantiationException, InterruptedException, NegativeArraySizeException, NullPointerException, NumberFormatException, RuntimeException, SecurityException, StringIndexOutOfBoundsException

Il définit les erreurs suivantes : Error, OutOfMemoryError, VirtualMachineError

86.2. Le package java.io

Il définit les interfaces suivantes : DataInput, DataOutput

Il définit les classes suivantes :

Nom	Rôle
ByteArrayInputStream	Lecture d'un flux d'octets bufférisé
ByteArrayOutputStream	Ecriture d'un flux d'octets bufférisé
DataInputStream	Lecture de données stockées au format Java
DataOutputStream	Ecriture de données stockées au format Aava
InputStream	Classe abstraite dont héritent toutes les classes gérant la lecture de flux par octets
InputStreamReader	Lecture d'octets sous la forme de caractères
OutputStream	Classe abstraite dont hérite toutes les classes gérant l'écriture de flux par octets
OutputStreamWriter	Ecriture de caractères sous la forme d'octets
PrintStream	
Reader	Classe abstraite dont héritent toutes les classes gérant la lecture de flux par caractères
Writer	Classe abstraite dont héritent toutes les classes gérant l'écriture de flux par caractères

Il définit les exceptions suivantes : EOFException, InterruptedIOException, IOException, UnsupportedEncodingException, UTFDataFormatException

86.3. Le package java.util

Il définit l'interface Enumeration

Il définit les classes suivantes :

Nom	Rôle
Calendar	Classe abstraite pour manipuler les éléments d'une date
Date	Classe qui encapsule une date
Hashtable	Classe qui encapsule une collection d'éléments composés d'une par paire clé/valeur
Random	Classe qui permet de générer des nombres aléatoires
Stack	Classe qui encapsule une collection de type pile LIFO
TimeZone	Classe qui encapsule un fuseau horaire
Vector	Classe qui encapsule une collection de type tableau dynamique

Il définit les exceptions EmptyStackException et NoSuchElementException

86.4. Le package javax.microedition.io

Il définit les interfaces suivantes :

Nom	Rôle
Connection	Interface pour une connexion générique
ContentConnection	
Datagram	Interface pour un paquet de données
DatagramConnection	Interface pour une connexion utilisant des paquets de données
InputConnection	Interface pour une connexion entrante
OutputConnection	Interface pour une connexion sortante
StreamConnection	Interface pour une connexion utilisant un flux
StreamConnectionNotifier	

87. MIDP

C'est le premier profile qui a été développé dont l'objectif principal est le développement d'application sur des machines aux ressources et à l'interface limitées tel qu'un téléphone cellulaire. Ce profil peut aussi être utilisé pour développer des applications sur des PDA de type Palm.

L'API du MIDP se compose des API du CDLC et de trois packages :

- javax.microedition.midlet : cycle de vie de l'application
- javax.microedition.lcdui : interface homme machine
- javax.microedition.rms : persistance des données

Des informations complémentaires et le téléchargement de l'implémentation de référence de ce profil peuvent être trouvées sur le site de Sun : http://java.sun.com/products/midp/

Il existe deux versions du MIDP :

- 1.0 : la dernière révision est la 1.0.3 dont les spécifications sont issues de la JSR 37
- 2.0 : c'est la version la plus récente dont les spécifications sont issues de la JSR 118

Ce chapitre contient plusieurs sections :

- ♦ Les Midlets
- ♦ L'interface utilisateur
- ♦ La gestion des événements
- ♦ Le stockage et la gestion des données
- ♦ Les suites de midlets
- ♦ Packager une midlet
- ♦ MIDP for Palm O.S.

87.1. Les Midlets

Les applications créées avec MIDP sont des midlets : ce sont des classes qui héritent de la classe abstraite javax.microedition.midlet.Midlet. Cette classe permet le dialogue entre le système et l'application.

Elle possède trois méthodes qui permettent de gérer le cycle de vie de l'application en fonction des trois états possibles (active, suspendue ou détruite) :

- startApp() : cette méthode est appelée à chaque démarrage ou redémarrage de l'application
- pauseApp() : cette méthode est appelée lors de la mise en pause de l'application
- destroyApp() : cette méthode est appelée lors de la destruction de l'application

Ces trois méthodes doivent obligatoirement être redéfinies.

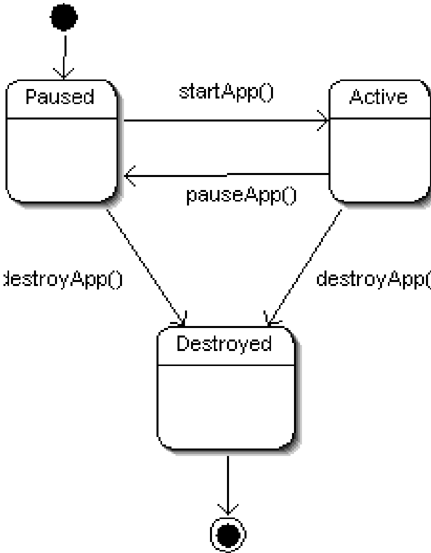
Exemple (MIDP 1.0) :

```
package perso.jmd.test.j2me;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class Test extends MIDlet {

  public Test() {
  }

  public void startApp() {
  }

  public void pauseApp() {
  }

  public void destroyApp(boolean unconditional) {
  }
}
```

Le cycle de vie d'une midlet est semblable à celui d'une applet. Elle possède plusieurs états :

- paused :
- active :
- destroyed :

Le changement de l'état de la midlet peut être provoqué par l'environnement d'exécution ou la midlet.



La méthode startApp() est appelée lors du démarrage ou redémarrage de la midlet. Il est important de comprendre que cette méthode est aussi appelée lors du redémarrage de la midlet : elle peut donc être appelée plusieurs fois au cours de l'exécution de la midlet.

La méthode pauseApp() est appelée lors de mise en pause de la midlet.

La méthode destroyApp() est appelée juste avant la destruction de la midlet.

87.2. L'interface utilisateur

Les possibilités concernant l'IHM de MIDP sont très réduites pour permettre une exécution sur un maximum de machines allant du téléphone portable au PDA. Ces machines sont naturellement et physiquement pourvues de contraintes forte concernant l'interface qu'ils proposent à leurs utilisateurs.

Avec le J2SE, deux API permettent le développement d'IHM : AWT et Swing. Ces deux API proposent des composants pour développer des interfaces graphiques riches de fonctionnalités avec un modèle de gestion des événements complet. Ils prennent en compte un système de pointage par souris, avec un écran couleur possédant de nombreuses couleurs et une résolution importante.

Avec MIDP, le nombre de composants et le modèle de gestion des événements sont spartiates. Il ne prend en compte qu'un écran tactile souvent monochrome ayant une résolution très faible. Avec un clavier limité en nombres de touches et dépourvu de système de pointage, la saisie de données sur de tels appareils est particulièrement limitée.

L'API pour les interfaces utilisateurs du MIDP est regroupée dans le package javax.microedition.lcdui.

Elle se compose des éléments de haut niveaux et des éléments de bas niveaux.

87.2.1. La classe Display

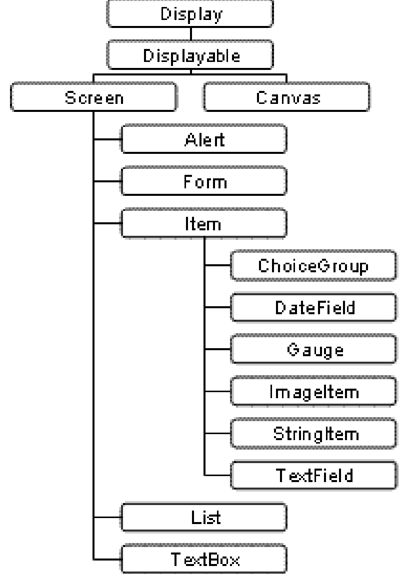
Pour pouvoir utiliser les éléments graphiques, il faut obligatoirement obtenir un objet qui encapsule l'écran. Un tel objet est du type de la classe Display. Cette classe possède des méthodes pour afficher les éléments graphiques.

La méthode statique getDisplay() renvoie une instance de la classe Display qui encapsule l'écran associé à la midlet fournie en paramètre de la méthode.

Exemple (MIDP 1.0) :

```
package perso.jmd.test.j2me;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class Hello extends MIDlet {

  private Display display;

  public Hello() {
    display = Display.getDisplay(this);
  }

  public void startApp() {
  }

  public void pauseApp() {
  }

  public void destroyApp(boolean unconditional) {
 }
}
```

Les éléments de l'interface graphique appartiennent à une hiérarchie d'objets : tous les éléments affichables héritent de la classe abstraite Displayable.



La classe Screen est la classe mère des éléments graphiques de haut niveau. La classe Canvas est la classe mère des éléments graphiques de bas niveau.

Il n'est pas possible d'ajouter directement un élément graphique dans un Display sans qu'il soit inclus dans un objet héritant de Displayable.

Un seul objet de type Displayable peut être affiché à la fois. La classe Display possède la méthode getCurrent() pour connaître l'objet courant affiché et la méthode setCurrent() pour afficher l'objet fourni en paramètre.

87.2.2. La classe TextBox

Ce composant permet de saisir du texte.

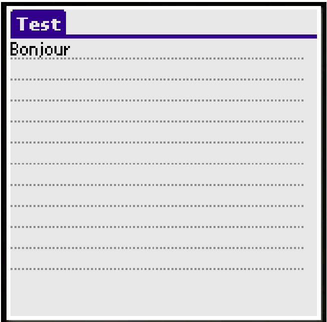
Exemple (MIDP 1.0) :

```
package perso.jmd.test.j2me;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class Hello extends MIDlet {

  private Display display;
  private TextBox textbox;

  public Hello() {
    display = Display.getDisplay(this);
    textbox = new TextBox("", "Bonjour", 20, 0);
  }

  public void startApp() {
    display.setCurrent(textbox);
  }

  public void pauseApp() {
  }

  public void destroyApp(boolean unconditional) {
  }
}
```

Résultat :

sur l'émulateur Palm

sur l'émulateur de téléphone mobile



87.2.3. La classe List

Ce composant permet la sélection d'un ou plusieurs éléments dans une liste d'éléments.

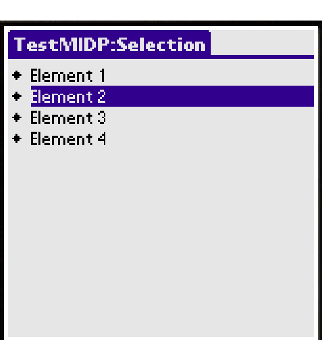
Exemple (MIDP 1.0) :

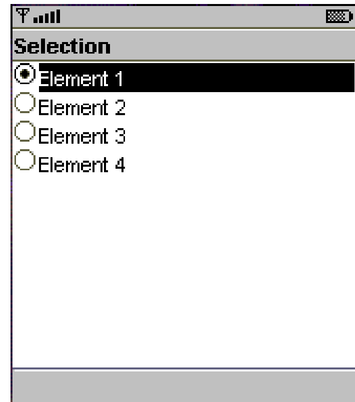
```
package perso.jmd.test.j2me;
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class Test extends MIDlet {
  private Display display;
  private List liste;

  protected static final String[] elements = {"Element 1",
                                             "Element 2",
                                             "Element 3",
                                             "Element 4"};

  public Test() {
    display = Display.getDisplay(this);
    liste = new List("Selection", List.EXCLUSIVE, elements, null);;
  }

  public void startApp() {
    display.setCurrent(liste);
  }

  public void pauseApp() {
  }

  public void destroyApp(boolean unconditional) {
  }
}
```

Résultat :

sur l'émulateur Palm

sur l'émulateur de téléphone mobile





La suite de cette section sera développée dans une version future de ce document

87.2.4. La classe Form

La classe From permet d'insérer dans l'élément graphique qu'elle représente d'autres éléments graphiques : cette classe sert de conteneurs. Les éléments insérés sont des objets qui héritent de la classe abstraite Item.

Exemple (MIDP 1.0) :

```
package perso.jmd.test.j2me;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class Hello extends MIDlet {

  private Display display;
  private Form mainScreen;

  public Hello() {
    display = Display.getDisplay(this);
  }

  public void startApp() {
    mainScreen = new Form("Hello");
    mainScreen.append("Bonjour");
   display.setCurrent(mainScreen);
  }

  public void pauseApp() {
  }

  public void destroyApp(boolean unconditional) {
  }
}
```

87.2.5. La classe Item

La classe javax.microedition.lcdui.Item est la classe mère de tous les composants graphiques qui peuvent être insérés dans un objet de type Form.

Cette classe définit seulement deux méthodes, getLabel() et setLabel() qui sont le getter et le setter pour la propriété label.

Il existe plusieurs composants qui héritent de la classe Item

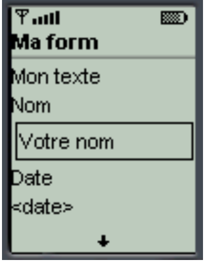
Classe	Rôle
ChoiceGroup	sélection d'un ou plusieurs éléments
DateField	affichage et saisie d'une date
Gauge	affichage d'une barre de progression
ImageItem	affichage d'une image
StringItem	affichage d'un texte
TextField	saisie d'un texte

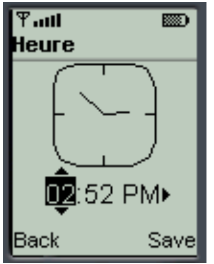
Exemple (MIDP 1.0) :

```
package perso.jmd.test.j2me;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class Hello extends MIDlet {

  private Display display;
  private Form form;
  private ChoiceGroup choiceGroup;
  private DateField dateField;
  private DateField timeField;
  private Gauge gauge;
  private StringItem stringItem;
  private TextField textField;

  public Hello() {
    display = Display.getDisplay(this);
    form = new Form("Ma form");

    String choix[] = {"Choix 1", "Choix 2"};
    stringItem = new StringItem(null,"Mon texte");
    choiceGroup = new ChoiceGroup("Sélectionner",Choice.EXCLUSIVE,choix,null);
    dateField = new DateField("Heure",DateField.TIME);
    timeField = new DateField("Date",DateField.DATE);
    gauge = new Gauge("Avancement",true,10,1);
    textField = new TextField("Nom","Votre nom",20,0);

    form.append(stringItem);
    form.append(choiceGroup);
    form.append(timeField);
    form.append(dateField);
    form.append(gauge);
    form.append(textField);
  }

  public void startApp() {
    display.setCurrent(form);
  }

  public void pauseApp() {
  }

  public void destroyApp(boolean unconditional) {
  }
```

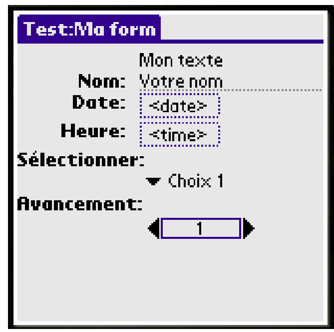
```
}
```

Résultat sur l'émulateur de téléphone mobile :





Résultat sur l'émulateur Palm OS :



87.2.6. La classe Alert

Cette classe permet d'afficher une boite de dialogue pendant un temps déterminé.

Elle possède deux constructeurs :

- un demandant le titre de l'objet
- un demandant le titre, le texte, l'image et le type de l'image

Elle possède des getters et des setters sur chacun de ces éléments.

Pour préciser le type de la boîte de dialogue, il faut utiliser une des constantes définies dans la classe AlertType dans le constructeur ou dans la méthode setType() :

Constante	type de la boite de dialogue
ALARM	informer l'utilisateur d'un événement programmé
CONFIRMATION	demander la confirmation à l'utilisateur
ERROR	informer l'utilisateur d'une erreur
INFO	informer l'utilisateur
WARNING	informer l'utilisateur d'un avertissement

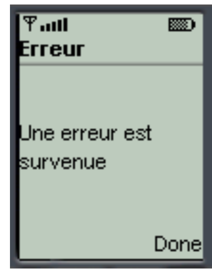
Pour afficher un objet de type Alert, il faut utiliser une version surchargée de la méthode setCurrent() de l'instance de la classe Display. Cette version nécessite deux paramètres : l'objet Alert à afficher et l'objet de type Displayable qui sera affiché lorsque l'objet Alert sera fermé.

La méthode setTimeout() qui attend un entier en paramètre permet de préciser la durée d'affichage en milliseconde de la boîte de dialogue. Pour la rendre modale, il faut lui passer le paramètre Alert.FOREVER.

Exemple (MIDP 1.0) :

```
package perso.jmd.test.j2me;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class Test extends MIDlet {

  private Display display;
private Alert alert;
  private Form form;

  public Test() {
    display = Display.getDisplay(this);
    form = new Form("Hello");
    form.append("Bonjour");

   alert = new Alert("Erreur", "Une erreur est survenue", null, AlertType.ERROR);
   alert.setTimeout(Alert.FOREVER);
  }

  public void startApp() {
    display.setCurrent(alert, form);
  }

  public void pauseApp() {
  }

  public void destroyApp(boolean unconditional) {
  }
}
```

Résultat sur l'émulateur de téléphone mobile:



Résultat sur l'émulateur Palm OS:

87.3. La gestion des événements

Les interactions entre l'utilisateur et l'application se concrétisent par le traitement d'événements particuliers pour chaque action.

MIDP définit interface de type Listener pour la gestion des événements :

Interface	Rôle
CommandListener	Listener pour une activation d'une commande
ItemStateListener	Listener pour un changement d'état d'un composant(modification du texte d'une zone de texte, ...)

Cette section sera développée dans une version future de ce document

87.4. Le stockage et la gestion des données

Avec MIDP, le mécanisme pour la persistance des données est appelé RMS (Record Management System). Il permet le stockage de données et leur accès ultérieur.

RMS propose un accès standardisé au système de stockage de la machine dans lequel s'exécute le programme. Il n'impose pas aux constructeurs la façon dont les données doivent être stockées physiquement.

Du fait de la simplicité des mécanismes utilisés, RMS ne définit qu'une seule classe : RecordStore. Cette classe ainsi que les interfaces et les exceptions qui composent RMS sont regroupées dans le package javax.microedition.rms.

Les données sont stockées dans un ensemble d'enregistrements (records). Un enregistrement est un tableau d'octets. Chaque enregistrement possède un identifiant unique nommé recordId qui permet de retrouver un enregistrement particulier.

A chaque fois qu'un ensemble de données est modifié (ajout, modification ou suppression d'un enregistrement), son numéro de version est incrémenté.

Un ensemble de données est associé à un unique ensemble composé d'une ou plusieurs Midlets (Midlet Suite).

Un ensemble de données possède un nom composé de 32 caractères maximum.

87.4.1. La classe RecordStore

L'accès aux données se fait obligatoirement en utilisant un objet de type RecordStore.

Les principales méthodes sont :

Méthode	Rôle
int addRecord(byte[],int, int)	Ajouter un nouvel enregistrement
void addRecordListener(RecordListener)	
void closeRecordStore()	Fermer l'ensemble d'enregistrement
void deleteRecord(int)	Supprimer l'enregistrement dont l'identifiant est fourni en paramètre
static void deleteRecordStore(String)	Supprimer l'ensemble d'enregistrements dont le nom est fourni en paramètre
Enumeration enumerateRecords(RecordFilter , RecordComparator, boolean)	Renvoyer une énumération pour parcourir tout ou partie de l'ensemble

String getName()	Renvoyer le nom de l'ensemble d'enregistrements
int getNextRecordID()	Renvoyer l'identifiant du prochain enregistrement créé
int getNumRecords()	Renvoyer le nombre d'enregistrement contenu dans l'ensemble
byte[] getRecord(int)	Renvoyer l'enregistrement dont l'identifiant est fourni en paramètre
int getRecord(int, byte[], int)	Obtenir les données contenues dans un enregistrement dont l'identifiant est fourni en paramètre. Renvoie le nombre d'octets de l'enregistrement
int getRecordSize(int)	Renvoyer la taille en octets de l'enregistrement dont l'identifiant est fourni en paramètre
int getSize()	Renvoyer la taille en octets occupée par l'ensemble
static String[] listRecordStores()	Renvoyer un tableau de chaîne de caractères contenant le nom des ensembles de données associées à l'ensemble de Midlet courant
static RecordStore openRecordStore(String, boolean)	Ouvrir un ensemble de données dont le nom est fourni en paramètre. Celui ci est créé s'il n'existe pas et que le booléen est à true
void setRecord(int, byte[], int, int)	Mettre à jour l'enregistrement précisé avec les données fournies en paramètre

Pour pouvoir utiliser un ensemble d'enregistrements, il faut utiliser la méthode statique openRecordStore() en fournissant le nom de l'ensemble et un booléen qui précise si l'ensemble doit être créé au cas où celui ci n'existe pas. Elle renvoie un objet RecordStore qui encapsule l'ensemble d'enregistrements.

L'appel de cette méthode peut lever l'exception RecordStoreNotFoundException si l'ensemble n'est pas trouvé, RecordStoreFullException si l'ensemble de données est plein ou RecordStoreException dans les autres cas problématiques.

La méthode closeRecordStore() permet de fermer un ensemble précédemment ouvert. Elle peut lever les exceptions RecordStoreNotOpenException et RecordStoreException.

La suite de cette section sera développée dans une version future de ce document

87.5. Les suites de midlets

Cette section sera développée dans une version future de ce document

87.6. Packager une midlet

Une application constituées d'une suite de midlets est packager sous la forme d'une archive .jar. Cette archive doit contenir un fichier manifest et tous les éléments nécessaire à l'exécution de l'application (fichiers .class et les ressources telles que les images, ...).

87.6.1. Le fichier manifest

Ce fichier contient des informations sur l'application.

Ce fichier contient une définition de propriétés utilisées par l'application. Ces propriétés sont sous la forme clé/valeur.

Plusieurs propriétés sont définis par les spécifications des midlets : celles-ci commencent par MIDLet-.

Propriétés	Rôle
MIDlet-Name	Nom de l'application
MIDlet-Version	Numéro de version de l'application
MIDlet-Vendor	Nom du fournisseur de l'application
MIDlet-Icon	Nom du fichier .png contenant l'icône de l'application
MIDlet-Description	Description de l'application
MIDlet-Info-URL	
MIDlet-Jar-URL	URL de téléchargement de fichier jar
MIDlet-Jar-Size	taille en octets du fichier .jar
MIDlet-Data-Profile	
MicroEdition-Configuration	

Il est possible de définir ces propres attributs

87.7. MIDP for Palm O.S.

MIDP for Palm O.S. est une implémentation particulière du profile MIPD pour le déploiement et l'exécution d'applications sur des machines de type Palm. Elle permet d'exécuter des applications écrites avec MIDP sur un PALM possédant une version 3.5 ou supérieure de cet O.S.

Cette implémentation remplace l'ancienne implémentation développée par Sun nommé KJava.

87.7.1. L'installation

MIPD for Palm O.S. peut être téléchargé à l'URL suivante : http://java.sun.com/products/midp4palm/download.html. Il faut télécharger le fichier midp4palm-1_0.zip et le fichier midp4palm-1_0-doc.zip qui contient la documentation.

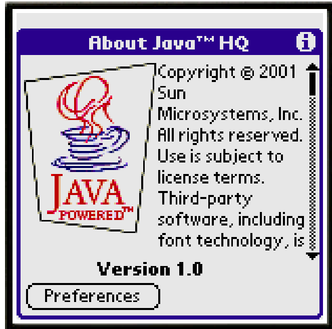
L'installation comprend une partie sur le poste de développement PC et une partie sur la machine Palm pour les tests d'exécution.

Pour pouvoir utiliser MIDP for Palm O.S., il faut déjà avoir installé CLDC et MIDP.

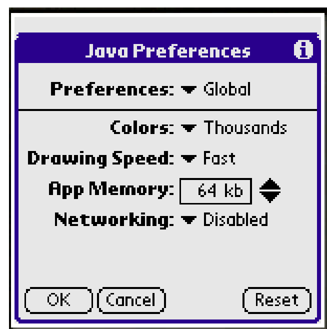
Il faut commencer l'installation sur le PC en décompressant les deux fichiers dans un répertoire.

Pour pouvoir exécuter les applications sur le Palm, il faut installer le fichier MIPD.prc contenu dans le répertoire PRCFiles sur le Palm en procédant comme pour toute application Palm.

En cliquant sur l'icône, on peut régler différents paramètres.



Un clic sur le bouton "Preferences" permet de modifier ces paramètres.



87.7.2. La création d'un fichier .prc

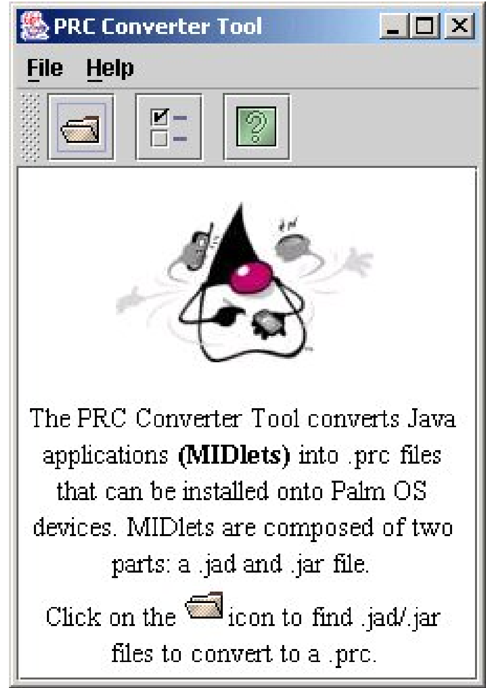
MIPD for Palm O.S. fourni un outil pour transformer les fichiers .jad et .jar qui composent une application J2ME en un fichier .prc directement installable sur un Palm.

Sous Windows, il suffit d'exécuter le programme converter.bat situé dans le sous répertoire Converter du répertoire d'installation.

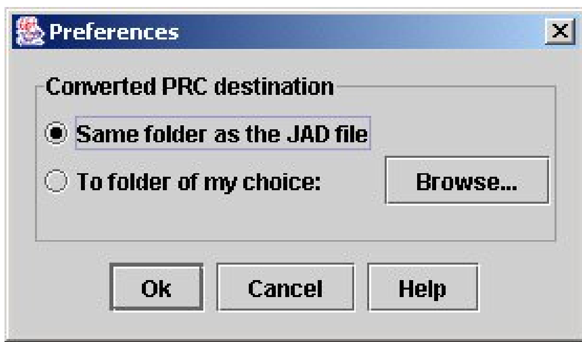
Il faut que la variable d'environnement JAVA_PATH pointe vers le répertoire d 'installation d'un JDK 1.3. minimum. Si ce n'est pas le cas, un message d'erreur est affiché.

```
Error: Java path is missing in your environment
         Please set JAVA_PATH to point to your Java directory
         e.g. set JAVA_PATH=c:\bin\jdk1.3\
```

Si tout est correct, l'application se lance.

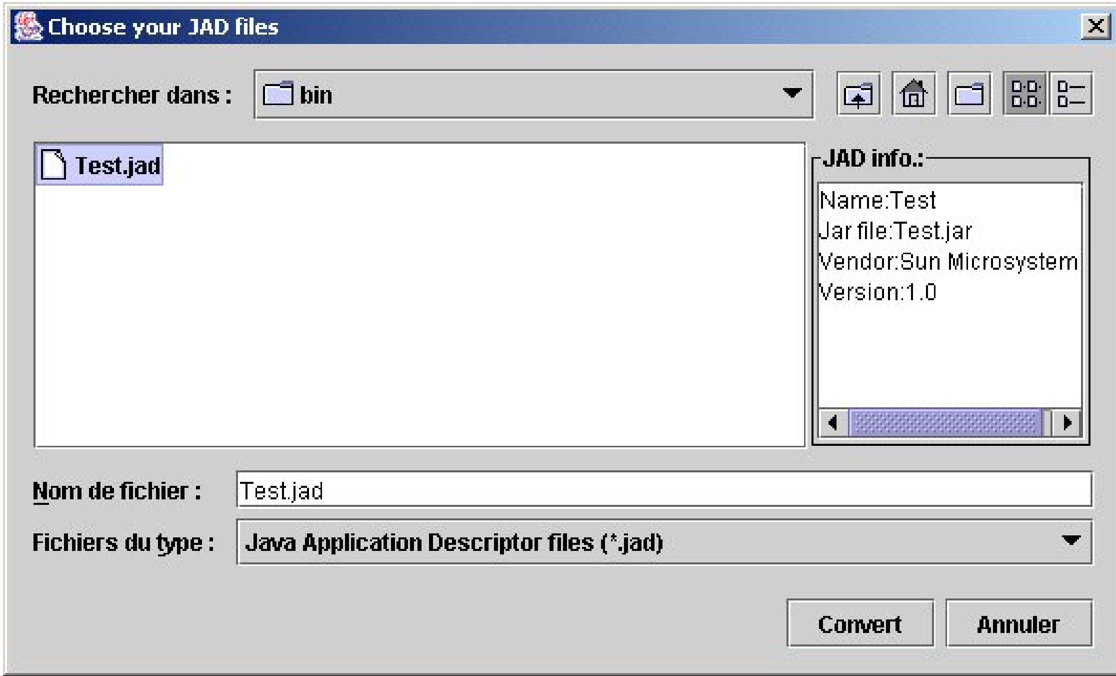


Il est possible de préciser le répertoire du ou des fichiers .prc généré en utilisant l'option "Preference" du menu "File" :



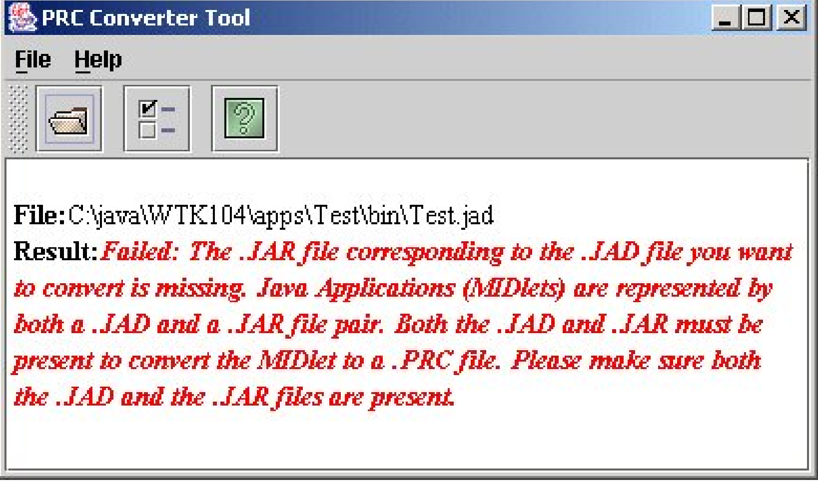
Une boîte de dialogue permet de choisir entre le même répertoire que celui qui contient le fichier .jad ou de sélectionner un répertoire quelconque.

Il suffit de cliquer sur l'icône en forme de répertoire dans la barre d'icônes pour sélectionner le fichier .jad. Les fichiers .jad et .jar de l'application doivent obligatoirement être ensemble dans le même répertoire.

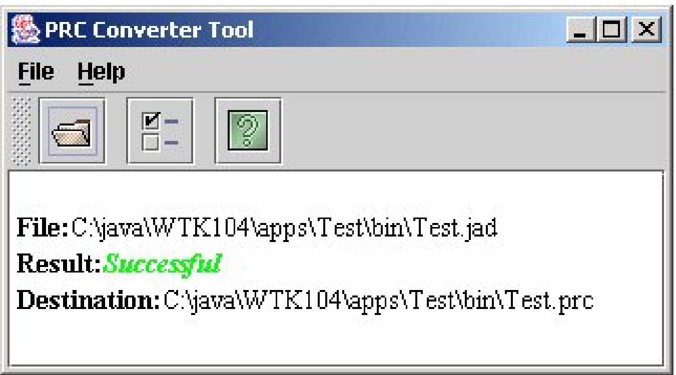


Un clic sur le bouton "Convert", lance la conversion.

Si la conversion échoue, un message d'erreur est affiché. Exemple, si le fichier .jar correspond au fichier .jad est absent, alors le message suivant est affiché :



Si toutes les opérations se sont correctement passées, alors un message récapitulatif est affiché :



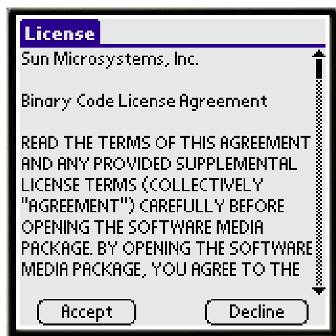
87.7.3. L'installation et l'exécution d'une application

Une fois le fichier .prc créé, il suffit d'utiliser la procédure standard d'installation d'un tel fichier sur le Palm (ajouter le fichier dans la liste avec "l'outil d'installation" du Palm et lancer une synchronisation).

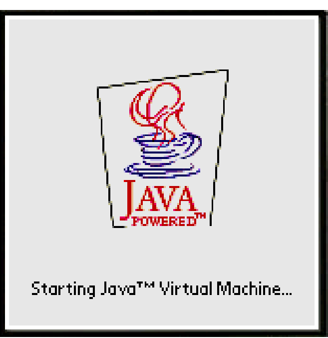
Une fois l'application installée, l'icône de l'application apparaît.

Pour exécuter l'application, il suffit comme pour une application native, de cliquer sur l'icône.

Lors de la première exécution, il faut lire et valider la licence d'utilisation.



Une splash screen s'affiche durant le lancement de la machine virtuelle.



Puis l'application s'exécute.

88. CDC

La suite de ce chapitre sera développée dans une version future de ce document

Cette configuration se destine à l'utilisation de Java sur des machines mobiles possédant un processeur 32 bits, au moins 2Mo de RAM et une connexion au réseau.

CDC est une spécification définit par la JSR numéro 036.

La machine virtuelle utilise par le CDC est nommé CVM. Elle respecte intégralement les spécifications de la plate-forme Java 2 version 1.3.

Le CDC ne peut être utilisé seul : il faut lui adjoindre un ou plusieurs profiles qui lui sont spécifiques.

Le CDC définit aussi un ensemble d'API de base :

- java.lang
- java.util
- java.net
- java.io
- java.text
- java.security

Le contenu de ces packages est très proche de celui de la plate-forme J2SE excepté quelques exceptions et surtout la suppression de toutes les API dépréciée (deprecated).

La version 1.1 du CDC est en cours de spécification dans la JSR 218

89. Les profils du CDC

La suite de ce chapitre sera développée dans une version future de ce document

Ce chapitre contient plusieurs sections :

- Foundation profile
- Le Personal Basis Profile (PBP)
- Le Personal Profile (PP)

89.1. Foundation profile

Ce profile sert de base pour le développement d'application sur des outils mobiles utilisant la configuration CDC tel que des Pockets PC ou des Tablets PC.

Il sert de base pour d'autres profiles.

Une partie importante de ce profil concerne les différentes formes de connexion au réseau.

89.2. Le Personal Basis Profile (PBP)

Ce profil contient les éléments de bases pour développer une interface graphique avec le CDC et le Foundation profil.

Ce profile a été développé sous la JSR 129.

89.3. Le Personal Profile (PP)

Ce profile se destine au développement d'applications sur des PDA disposant de ressources importantes tel que les Pockets PC. Ce profil permet notamment le développement d'IHM évoluée.

Le Personal Basis Profile est un sous ensemble du Personal Profile.

http://java.sun.com/products/personalprofile/

Ce profile a été développé sous la JSR 62.

90. Les autres technologies pour les applications mobiles

La suite de ce chapitre sera développée dans une version future de ce document

Ce chapitre contient plusieurs sections :

- KJava
- PDAP (PDA Profile)
- PersonalJava
- Java Phone
- JavaCard
- Embedded Java
- Waba, Super Waba, Visual Waba

90.1. KJava

Ce n'est pas un profil officiel mais un développement proposé par Sun pour réaliser des développements sur des machines de type Palm.

KJava n'est plus supporté par Sun. Il faut utiliser MIDP for Palm O.S. à la place.

90.2. PDAP (PDA Profile)

Ce profile permet le développement d'application sur PDA en tenant compte notamment de l'accès aux données. Il utilise la configuration CLDC.

Il propose deux packages optionnels :

- Personal Information Management (PIM) : pour standardiser l'accès aux données personnelles stockées dans la plupart des PDA tels que le carnet d'adresse, l'agenda, le bloc-notes, ...
- File Connection (FC) : pour permettre l'accès aux données stockées dans un système de fichiers externe tel que les cartes mémoires

Les spécifications de ce profile sont en cours de développement sous la JSR 075 : http://jcp.org/en/jsr/detail?id=075 (PDA Optional Packages for the J2ME Platform)

90.3. PersonalJava

http://java.sun.com/products/personaljava/

La dernière version de ces spécifications est la 1.2.

PersonalJava est composé de packages obligatoires et facultatifs.

Sun propose un outil pour émuler un environnement d'exécution pour des applications développer avec PersonalJava : PJEE (PersonalJava Emulation Environment).

Ce profile a été abandonné au profit d'un ensemble de profils qui respecte mieux le découpage des rôles de J2ME : CDC, Foundation profile et Personal Profile.

90.4. Java Phone

http://java.sun.com/products/javaphone/

90.5. JavaCard

http://java.sun.com/products/javacard/

90.6. Embedded Java

Cette technologie n'est plus supportée par Sun qui propose en remplacement CLDC et MIDP de la plate-forme J2ME.

http://java.sun.com/products/embeddedjava/

90.7. Waba, Super Waba, Visual Waba

Partie 13 : Annexes

Annexe A : GNU Free Documentation License

Version 1.1, March 2000

Copyright (C) 2000 Free Software Foundation, Inc.
59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND défіniTIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text

editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative définition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

Annexe B : Glossaire

A	B	C	D	E	F
G	H	I	J	K	L
M	N	O	P	Q	R
S	T	U	V	W	X
Y	Z				

A

API (Application Programming Interface)	Une API est une bibliothèque qui regroupe des fonctions sous forme de classes pouvant être utilisées pour développer.
Applet	C'est une petite application java compilée, incluse dans une page html, qui est chargée par un navigateur et qui est exécutée sous le contrôle de celui ci. Pour des raisons de sécurité, par défaut, les applets ont des possibilités très resteintes.
AWT (Abstract Window Toolkit)	C'est une bibliothèque qui regroupe des classes pour développer des interfaces graphiques. Ces composants sont dit "lourds" car ils utilisent les composants du système sur lequel ils s'executent. Ainsi, le nombre des composants est volontairement resteints pour ne conserver que les composants présents sur tous les systèmes.

B

BDK (Beans Development Kit)	C'est un outil fourni par Sun qui permet d'assembler des beans de façon graphique pour générer des applications.
Bean	C'est un composant réutilisable. Il possède souvent une interface graphique mais pas obligatoirement.
BluePrints	Ce sont des documents proposés par Sun pour faciliter le développement avec Java (exemple de code, conseils, design patterns, FAQ, ...)
BMP (Bean Managed Persistence)	Type d''EJB entité dont la persistance est à la charge du code qu'il contient
Byte code	Un programme source java est compilé en byte code. C'est un langage machine indépendant du processeur. Le byte code est ensuite traduit par la machine virtuelle en langage machine compréhensible par le système ou il s'execute. Ceci permet de rendre java indépendant de tout système.

C

CLASSPATH	C'est une variable d'environement qui contient les répertoires contenant des bibliothèques utilisables pour la compilation et l'execution du code.

CLDC (Connected Limited Device Configuration)	Configuration J2ME pour des appareils possédant de faibles ressources et une interface utilisateur réduite tels que des téléphones mobiles, des pagers, des PDA, etc ...
CDC (Connected Device Configuration)	Configuration J2ME pour des appareils embarqués possédant certaines ressources et une connexion à internet tels que des set top box, certains PDA haut de gamme, des systèmes de navigations pour voiture, etc ...
CMP (Container Managed Persistence)	Type d'EJB entité dont la persistance est assurée par le conteneur
CORBA (Common Object Request Broker Architecture)	C'est un modèle d'objets distribués indépendant du langage de développement des objets dont les spécifications sont fournies par l'OMG.
Core class	C'est une classe standard qui est disponible sur tous les systèmes ou tourne Java.
Core packages	C'est l'ensemble des packages qui composent les API de la plate-forme Java.

D

Deprecated	Terme anglais qui peut être attribué une classe, une interface, un constructeur, une méthode ou un attribut lorque celle ci ne doivent plus être utilisés car Sun ne garantit pas que cet élément sera encore présent dans les prochaines versions de l'API.
DOM (Document Object Model)	Spécification et API pour représenter et parcourir un document XML sous la forme d'un arbre en mémoire

E

EAR (Enterprise ARchive)	Archive qui contient une application J2EE
EJB (Entreprise Java Bean)	Les EJB sont des composants métier qui répondent à des spécifications précises. Il existe deux types d'EJB : EJB Entity qui s'occupe de la persistence des données et EJB session qui gère les traitements. Les EJB doivent s'executer sur un serveur dans un conteneur d'EJB.
Exception	C'est un mécanisme qui permet de gérer les anomalies et les erreurs détectées dans une application en facilitant leur détections et leur traitement. Les exceptions sont largement utilisées et intégrées dans le langage java pour accroitre la sécurité du code.

F

G

Garbage Collector (Ramasse miettes)	C'est un mécanisme intégré à la machine virtuelle qui récupère automatiquement la mémoire inutilisée en restituant les zones de mémoire laissées libres suite à la destruction des objets.

H

HotJava	Navigateur web de Sun écrit en Java
HTML (HyperText Markup Language)	Langage à base de balises pour formatter une page web affichée dans un navigateur

I

IDL (Interface définition Language)	Langage qui permet de définir des objets devant être utilisé avec CORBA
IIOP (Internet Iter Orb Protocole)	Protocole pour faire communiquer des objets CORBA
Interface	C'est une définition de méthodes et de variables de classes que doivent respecter les classes qui l'implémente. Une classe peut implémenter plusieurs interfaces. La classe doit définir toutes les méthodes des interfaces sinon elle est abstraite.
Introspection	Fonction qui permet d'obtenir dynamiquement les entités (champs et méthodes) qui composent un objet

J

J2EE (Java 2 Entreprise Edition)	C'est une version du JDK qui contient la version standard plus un ensemble de plusieurs API permettant le développement d'applications destinées aux entreprises : EJB, Servlet, JSP, JNDI, JMS, JTA, JTS, ...
J2ME (Java 2 Micro Edition)	C'est une version du JDK qui contient le nécessaire pour développer des applications capables de fonctionner dans des environnements limités tels que les assistants personnels (PDA), les téléphones portables ou les systèmes de navigation embarqués
J2SE (Java 2 Standard Edition)	C'est une version du JDK qui contient le nécessaire pour développer des applications et des applets.
JAAS (Java Authentication and Authorization Service)	API qui permet d'authentifier un utilisateur et de leur accorder des droits d'accès
JAI (Java Advanced Imaging)	API dédié à l'utilisation et à la transformation d'images
JAR (Java ARchive)	Technique qui permet d'archiver avec ou sans compression des classes java et des ressources dans un fichier unique de façon indépendante de toute plate-forme. Ce format supporte aussi la signature électronique.
Java Media API	Regroupement d'API pour le multimédia
Java One	Conférence des développeurs Java périodiquement organisée par Sun
JavaHelp	Système d'aide pour les utilisateurs d'application entièrement écrit en java
JavaMail	API pour utiliser la messagerie électronique (e mail)
Java Web Start	Outil qui permet d'utiliser une application client par téléchargement automatique via le réseau
Java XML Pack	Regroupe des API pour l'utilisation de XML avec Java
JAXB (Java API for XML Binding)	API pour faciliter la persistence entre objets java et document XML

JAXM (Java API for XML Messaging)	API pour échanger des messages XML notamment avec les services web
JAXP (Java API for XML Processing)	API pour parcourir un document XML (DOM et SAX) et le transformer avec XSLT
JAXR (Java API for XML Registries)	API pour utiliser les services d'annuaires pour les services web (UDDI)
JAX-RPC (Java API for XML Remote Procedure Calls)	API pour utiliser l'appelle de méthodes distantes via SOAP
JCA (Java Connector Architecture)	Spécification pour normaliser le développement de connecteurs vers des progiciels
JCP (Java Community Process)	Processus utilisé par Sun et de nombreux partenaires pour gèrer les évolutions de java et de ces API
JDBC (Java Data Base Connectivity)	C'est une API qui permet un accès à des bases de données tout en restant indépendante de celles ci. Un driver spécifique à la base utilisée permet d'assurer cette indépendance car le code Java reste le même.
JDC (Java Delevopper Connection)	C'est un service en ligne proposé gratuitement par Sun après enregistrement qui propose de nombreuses ressources sur java (tutorial, cours, information, mailing ...).
JDO (Java Data Objects)	API et spécification pour faciliter le mapping entre objet java et une source de données
JDK (Java Development Kit)	C'est l'environnement de développement Java.Il existe plusieurs versions majeures : 1.0, 1.1, 1.2 (aussi appelée Java 2) et 1.3. Tous les outils fournis sont à utiliser avec une ligne de commandes.
JFC (Java Foundation Class)	C'est un ensemble de classes qui permet de développer des interfaces graphiques plus riches et plus complets qu'avec AWT
JIT Compiler (Just In Time Compiler)	C'est un compiltateur qui compile le byte-code à la volée lors de l'exécution des programme pour améliorer les performances.
JMS (Java Messaging Service)	C'est une API qui permet l'échange de messages assynchrones entre applications en utilisant un MOM (Middleware Oriented Message)
JMX (Java Management eXtension)	API et spécification qui permet de développer un système d'administration d'application à distance via le réseau
JNDI (Java Naming and Directory Interface)	C'est une bibliothèque qui permet un accès aux annuaires de l'entreprise. Plusieurs protocoles sont supportés : LDAP, DNS, NIS et NDS.
JNI (Java Native Interface)	C'est un API qui normalise et permet les appels de code natif dans une application java.
JRE (Java Runtime Environment)	C'est l'environnement d'execution des programmes Java.
JSDK (Java Servlet Development Kit)	C'est un ensemble de deux packages qui permettent le développement des servlets.
JSP (Java Server Page)	C'est une technologie comparable aux ASP de Microsoft mais utilisant Java. C'est une page HTML enrichie de tag JSP et de code Java. Une JSP est traduite en servlet pour être executée. Ceci permet de séparer la logique de présentation et la logique de traitement contenu dans un composant serveur tel que des servlets, des EJB ou des beans.
JSR (Java Specification Request)	Démande d'évolution ou d'ajout des API Java traité par le JCP
JSSE (Java Secure Socket Extension)	API permettant l'utilisation du protocole SSL pour des échange HTTP sécurisé
JSTL (Java Standard Tag Library)	Bibliothèque de tags JSP standard

JTS (Java Transaction Service)	API pour utiliser les transactions
JTWI	Spécification issue de la JSR 185 visant à définir un environnement d'exécution utilisant CLDC, MIDP et plusieurs profiles de façon homogène
JUG (Java User Group)	Groupe d'utilisateurs Java
JVM (Java Virtual Machine)	C'est la machine virtuelle dans laquelle s'éxecute le code Java. C'est une application native dépendante du système d'exploitation sur laquelle elle s'execute. Elle répond à des normes dictées par Sun pour assurer la portabilité du langage. Il en existe plusieurs développées par plusieurs éditeurs notamment Sun, IBM, Borland, Microsoft, ...

K

L

Layout Manager (gestionnaire de présentation)	Les layout manager sont des classes qui gèrent la disposition des composants d'une interface graphique sans utiliser des coordonnées.
LDAP	Protocole qui permet d'accéder à un annuaire d'entreprise

M

Message Driven Bean	Type d'EJB qui traite les messages reçus d'un MOM de façon assynchrone
Midlet	Application mobile développée avec CLDC et MIDP
MIDP (Mobile Information Device Profile)	Profil utilisé avec la configuration CLDC pour le développement d'applications mobiles sous la forme de Midlets
MOM (Middleware Oriented Message)	Outil qui permet l'échange de messages entre applications
MVC (Model View Controler)	Modèle de conception largement répandu qui permet de séparer l'interface graphique, les traitements et les données manipulées

N

O

ODBC (Open Database Connectivity)	API et spécifications de Microsoft pour l'accès aux bases de données sous Windows
ORB	Middleware orienté objet pour mettre en oeuvre CORBA

P

Package (Paquetage)	Il permettent des regrouper des classes par critères. Ils impliquent une structuration des classes dans une arborescence correspondant au nom donné au package.

Q

R

Ramasse miette	
RI (Reference Implementation)	Implementation de référence proposée par Sun d'une spécification particulière
RMI (Remote Method Invocation)	C'est une technologie développée par Sun qui permet de faire des appels d'objets distants. Cette technologie est plus facile à mettre en oeuvre que Corba mais elle ne peut appeler que des objets java.

S

Sandbox (bac à sable)	Il désigne un ensemble des fonctionnalités et d'objets qui assure la sécurité des applications. Son composant principal est le gestionnaire de sécurité. Par exemple, Il empeche par défaut à une applet d'accéder aux ressources du système.
SAX (Simple API for XML)	API pour traiter séquentiellement un document XML en utilisant des événements
Serialization	Fonction qui permet à un objet d'envoyer son état dans un flux pour permettre sa persistance ou son envoi à travers un reseau par exemple
Servlet	C'est un composant Java qui s'exécute côté serveur dans un environnement dédié pour répondre à des requêtes. L'usage le plus fréquent est la génération dynamique de page Web. On les compare souvent aux applets qui s'éxecutent côté client mais elles n'ont pas d'interface graphique.
SOAP (Simple Object Access Protocol)	Protocole des services web pour l'échange de messages et l'appel de méthodes distances grace à XML
SQL/J	spécification qui permet d'imbriquer du code SQL dans du code Java
Swing	Framework pour le développement d'interfaces graphiques composé de composants légers

T

Taglibs	Bibliothèques de tags personnalisés utilisés dans les JSP

U

V

W

WAR (Web ARchive)	Archive qui contient une application web

webapp (web application)	Application web reposant sur les servlets et les JSP

X

Y

Z