2

Introduction au framework Spring

Au sommaire de ce chapitre

- Le framework Spring
- Installer le framework Spring
- Configurer un projet Spring
- Installer Spring IDE
- Utiliser les outils de gestion des beans de Spring IDE
- En résumé

Ce chapitre est une vue d'ensemble du framework Spring, dont l'architecture s'articule autour de multiples modules organisés de manière hiérarchique. Vous allez faire connaissance avec les principales fonctions de chaque module et aurez un aperçu des éléments les plus importants de Spring 2.0 et 2.5. Spring est non seulement un framework applicatif, mais également une plate-forme qui héberge plusieurs projets connexes regroupés sous le nom Spring Portfolio. Dans ce chapitre, vous aurez un aperçu de leurs caractéristiques.

Avant de commencer à utiliser Spring, vous devez l'installer sur votre machine de développement. L'installation du framework est très simple. Toutefois, pour tirer le meilleur parti de Spring, vous devez comprendre la structure de son répertoire d'installation et le contenu de chaque sous-répertoire.

L'équipe de développement de Spring a créé un plug-in Eclipse, appelé Spring IDE, pour faciliter le développement d'applications Spring. Vous apprendrez à installer cet IDE et à utiliser ses fonctionnalités de gestion des beans.

À la fin de ce chapitre, vous aurez acquis de solides connaissances concernant l'architecture globale et les principales fonctionnalités de Spring. Vous saurez également installer le framework Spring et le plug-in Spring IDE sur votre machine.

2.1 Le framework Spring

Spring (**http://www.springframework.org/**) est un framework complet d'applications Java/Java EE hébergé par SpringSource (**http://www.springsource.com/**), connu précédemment sous le nom Interface21. Spring prend en charge de nombreux aspects du développement d'une application Java/Java EE et peut vous aider à produire plus rapidement des applications de qualité et de performances élevées.

Le cœur du framework Spring est constitué d'un conteneur IoC léger capable d'ajouter de manière déclarative des services d'entreprise à de simples objets Java. Spring s'appuie énormément sur une excellente méthodologie de programmation, la programmation orientée aspect (POA), pour apporter ces services aux composants. Dans le contexte du conteneur Spring IoC, les composants sont également appelés beans.

Le framework Spring lui-même se fonde sur de nombreux design patterns, notamment les patterns orientés objets du GoF (*Gang of Four*, le Gang des quatre) et ceux de Core Java EE de Sun. En utilisant Spring, nous sommes conduits à employer les meilleures pratiques industrielles pour concevoir et implémenter nos applications.

Spring n'entre pas en concurrence avec les technologies existantes dans certains domaines. Bien au contraire, il s'intègre avec de nombreuses technologies majeures afin de simplifier leur utilisation. Spring représente ainsi une bonne solution parfaitement adaptée à de nombreuses situations.

Les modules de Spring

L'architecture du framework Spring repose sur des modules (voir Figure 2.1). La flexibilité d'assemblage de ces modules est telle que les applications peuvent s'appuyer sur différents sous-ensembles de manière variée.

À la Figure 2.1, les modules sont présentés sous forme hiérarchique, les modules supérieurs dépendant des modules inférieurs. Le module Core se trouve tout en bas car il constitue les fondations du framework Spring.

- **Core.** Ce module fournit les fonctionnalités de base de Spring. Il propose un conteneur IoC de base nommé `BeanFactory`. Les fonctionnalités de ce conteneur seront décrites au Chapitre 3.
- **Context.** Ce module se fonde sur le module Core. Il en étend les fonctionnalités et propose un conteneur IoC élaboré, nommé `ApplicationContext`, qui ajoute ses

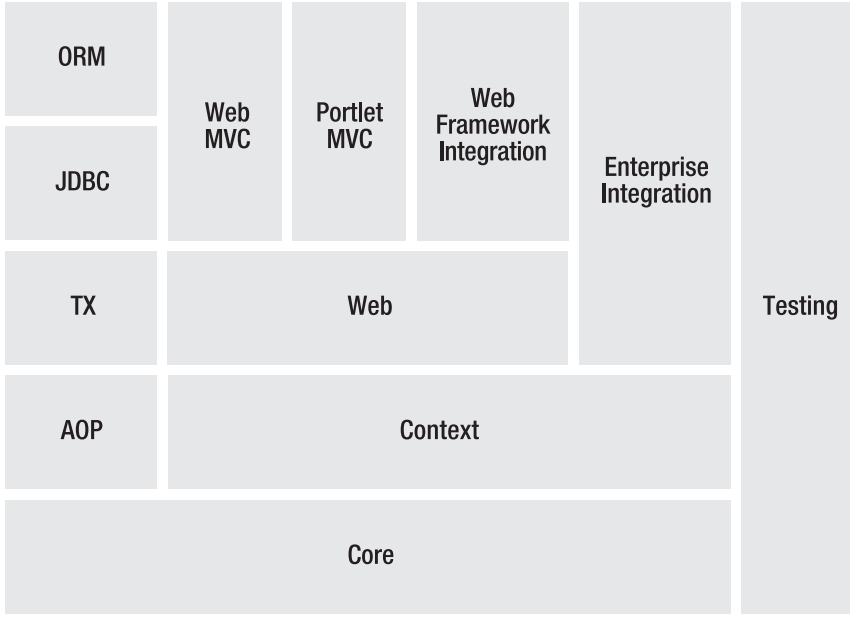


Figure 2.1

Un aperçu des modules du framework Spring.

propres fonctionnalités, comme la prise en charge de l'internationalisation (I18N), la communication à base d'événements et le chargement de ressources. Les caractéristiques de ce conteneur IoC élaboré seront examinées au Chapitre 4.

- **AOP.** Ce module définit un framework de programmation orientée aspect appelé Spring AOP. Avec l'IoC, la programmation orientée aspect est un autre concept fondamental de Spring. Les Chapitres 5 et 6 présenteront les approches POA classiques et nouvelles de Spring, ainsi que l'intégration d'AspectJ avec Spring.
- **JDBC.** Ce module définit au-dessus de l'API JDBC native une couche abstraite qui permet d'employer JDBC avec des templates et donc d'éviter la répétition du code standard (*boilerplate code*). Il convertit également les codes d'erreur des fournisseurs de bases de données en une hiérarchie d'exceptions `DataAccessException` propres à Spring. Les détails de la prise en charge de JDBC par Spring seront donnés au Chapitre 7.
- **TX.** Ce module permet de gérer les transactions par programmation ou sous forme déclarative. Ces deux approches peuvent être employées pour ajouter des possibilités transactionnelles aux objets Java simples. La gestion des transactions sera étudiée au Chapitre 8.

- **ORM.** Ce module intègre à Spring les frameworks classiques de correspondance objet-relationnel, comme Hibernate, JDO, TopLink, iBATIS et JPA. Tous les détails de l'intégration ORM de Spring se trouvent au Chapitre 9.
- **Web MVC.** Ce module définit un framework d'applications web conforme au design pattern Modèle-Vue-Contrôleur (MVC). Ce framework s'appuie sur les fonctionnalités de Spring pour qu'elles puissent servir au développement des applications web. Il sera examiné au Chapitre 10.
- **Web Framework Integration.** Ce module facilite l'utilisation de Spring en tant que support à d'autres frameworks web répandus, comme Struts, JSF, WebWork et Tapestry. Le Chapitre 11 s'intéressera à l'association de Spring avec plusieurs frameworks web.
- **Testing.** Ce module apporte la prise en charge des tests unitaires et des tests d'intégration. Il définit le framework Spring TestContext, qui rend abstraits les environnements de test sous-jacents comme JUnit 3.8, JUnit 4.4 et TestNG. Le test des applications Spring sera étudié au Chapitre 12.
- **Portlet MVC.** Ce module définit un framework de portlet, également conforme au design pattern MVC.
- **Enterprise Integration.** Ce module intègre à Spring de nombreux services d'entreprise répandus, notamment des technologies d'accès distant, les EJB, JMS, JMX, le courrier électronique et la planification, afin d'en faciliter l'usage.

Les versions de Spring

Deux ans et demi après la sortie du framework Spring 1.0 en mars 2004, la première mise à jour importante, Spring 2.0, a été publiée en octobre 2006. Voici ses principales améliorations et nouvelles fonctionnalités :

- **Configuration à base de schéma XML.** Dans Spring 1.x, les fichiers XML de configuration des beans n'acceptent que des DTD et toutes les définitions de caractéristiques doivent se faire au travers de l'élément `<bean>`. Spring 2.0 prend en charge la configuration à base de schéma XML, qui permet d'utiliser les nouvelles balises de Spring. Les fichiers de configuration des beans peuvent ainsi être plus simples et plus clairs. Dans ce livre, nous exploitons cette possibilité. Le Chapitre 3 en présentera les bases.
- **Configuration à base d'annotations.** En complément de la configuration basée sur XML, Spring 2.0 accepte dans certains modules une configuration à base d'annotations, comme `@Required`, `@Transactional`, `@PersistenceContext` et `@PersistenceUnit`. Les Chapitres 3, 8 et 9 expliqueront comment employer ces annotations.

- **Nouvelle approche de Spring AOP.** L'utilisation classique de la programmation orientée aspect dans Spring 1.x se fait au travers d'un ensemble d'API propriétaires de Spring AOP. Spring 2.0 propose une toute nouvelle approche fondée sur l'écriture de POJO avec des annotations AspectJ ou une configuration à base de schéma XML. Le Chapitre 6 en donnera tous les détails.
- **Déclaration plus aisée des transactions.** Dans Spring 2.0, il est beaucoup plus facile de déclarer des transactions. La nouvelle approche de Spring AOP permet de déclarer des greffons (*advices*) transactionnels, mais il est également possible d'appliquer des annotations `@Transactional` avec la balise `<tx:annotation-driven>`. Le Chapitre 8 reviendra en détail sur la gestion des transactions.
- **Prise en charge de JPA.** Spring 2.0 ajoute la prise en charge de l'API de persistance Java dans son module ORM. Elle sera examinée au Chapitre 9.
- **Bibliothèque pour la balise `form`.** Spring 2.0 propose une nouvelle bibliothèque pour faciliter le développement de formulaires dans Spring MVC. Son utilisation sera décrite au Chapitre 10.
- **Prise en charge de JMS en mode asynchrone.** Spring 1.x n'accepte que la réception synchrone de messages JMS par l'intermédiaire de `JmsTemplate`. Spring 2.0 ajoute la réception asynchrone de messages JMS au travers de POJO orientés message.
- **Prise en charge d'un langage de script.** Spring 2.0 reconnaît l'implémentation de beans à l'aide des langages de script JRuby, Groovy et BeanShell.

Novembre 2007 a vu la sortie de Spring 2.5, qui ajoute les fonctionnalités suivantes par rapport à Spring 2.0. Au moment de l'écriture de ces lignes, Spring 2.5 est la version courante du framework.

- **Configuration à base d'annotations.** Spring 2.0 a ajouté plusieurs annotations pour simplifier la configuration des beans. Spring 2.5 en reconnaît d'autres, notamment `@Autowired` et celles de la JSR 250, `@Resource`, `@PostConstruct` et `@PreDestroy`. Les Chapitres 3 et 4 expliqueront comment les utiliser.
- **Scan de composants.** Spring 2.5 peut détecter automatiquement les composants avec des annotations particulières situés dans le chemin d'accès aux classes, sans aucune configuration manuelle. Cette possibilité sera étudiée au Chapitre 3.
- **Prise en charge du tissage AspectJ au chargement.** Spring 2.5 prend en charge le tissage des aspects AspectJ dans le conteneur Spring IoC au moment du chargement. Il est ainsi possible d'utiliser des aspects AspectJ en dehors de Spring AOP. Ce sujet sera examiné au Chapitre 6.

- **Contrôleurs web à base d'annotations.** Spring 2.5 propose une nouvelle approche basée sur les annotations pour le développement des contrôleurs web. Il peut détecter automatiquement les classes contrôleurs avec l'annotation `@Controller`, ainsi que les informations configurées à l'aide de `@RequestMapping`, `@RequestParam` et `@ModelAttribute`. Le Chapitre 10 reviendra en détail sur ce point.
- **Prise en charge améliorée des tests.** Spring 2.5 définit un nouveau framework de test appelé Spring TestContext. Il prend en charge les tests dirigés par les annotations et rend abstraits les frameworks de test sous-jacents. Ce sujet fera l'objet du Chapitre 12.

Sachez enfin que le framework Spring est conçu de manière rétrocompatible et qu'il est donc facile de passer les applications Spring 1.x à Spring 2.0, ainsi que les applications 2.0 à 2.5.

Spring 3.0 est en préparation et, au moment de l'écriture de ces lignes, la version M2 (milestone 2) vient d'être publiée. Cette nouvelle version du framework poursuit ce qui avait débuté avec Spring 2.5, c'est-à-dire l'adhésion au modèle de programmation de Java 5. La liste officielle des nouvelles fonctionnalités n'est pas établie mais devrait inclure les éléments suivants :

- Généralisation de Spring EL : il sera possible d'utiliser les Expression Language dans n'importe quel module de Spring.
- Prise en charge des Portlets 2.0.
- Validation du modèle *via* les annotations.
- Intégration de la portée `conversation` dans Spring Core (elle est déjà présente dans Spring WebFlow).

La rétrocompatibilité de Spring 3.0 ne devrait être assurée qu'avec Spring 2.5.

Les projets Spring

Spring n'est pas seulement un framework applicatif. Il sert également de plate-forme à plusieurs projets open-source qui se fondent sur le projet central Spring Framework. Au moment de l'écriture de ces lignes, voici les projets de Spring Portfolio :

- **Spring IDE.** Ce projet fournit des plug-in Eclipse qui facilitent l'écriture des fichiers de configuration des beans. Depuis la version 2.0, Spring IDE prend également en charge Spring AOP et Spring Web Flow. Nous verrons comment installer Spring IDE dans ce chapitre.

- **Spring Security.** Ce projet, précédemment nommé Acegi Security, définit un framework de sécurité pour les applications d'entreprise, plus particulièrement celles développées avec Spring. Il propose des options de sécurité pour l'authentification, l'autorisation et le contrôle d'accès, que nous pouvons appliquer à nos applications.
- **Spring Web Flow.** Ce projet nous permet de modéliser sous forme de flux les actions complexes des utilisateurs au sein d'une application web. Grâce à Spring Web Flow, nous pouvons développer facilement ces flux web et les réutiliser.
- **Spring Web Services.** Ce projet se concentre sur le développement de services web roientés contrat (*contract-first*) ou pilotés par les documents (*document-driven*). Il intègre de nombreuses méthodes de manipulation d'un contenu XML.
- **Spring Rich Client.** Ce projet définit un framework, construit au-dessus de Spring, pour le développement d'applications graphiques riches avec Spring.
- **Spring Batch.** Ce projet propose un framework pour le traitement par lots dans les applications d'entreprise, en se focalisant sur les grands volumes d'informations.
- **Spring Modules.** Ce projet intègre d'autres outils et projets sous forme de modules afin d'étendre le framework Spring sans développer le module Core.
- **Spring Dynamic Modules.** Ce projet prend en charge la création d'applications Spring qui s'exécutent sur la plate-forme de services OSGi (*Open Services Gateway initiative*). Elle permet d'installer, de mettre à jour et de supprimer dynamiquement des modules applicatifs.
- **Spring Integration.** Ce projet fournit une extension du framework Spring qui prend en charge l'intégration avec des systèmes externes au travers d'adaptateurs de haut niveau.
- **Spring LDAP.** Ce projet propose une bibliothèque qui simplifie les opérations LDAP et gère les exceptions LDAP par une approche fondée sur les templates.
- **Spring JavaConfig.** Ce projet apporte une alternative Java à la configuration des composants dans le conteneur Spring IoC.
- **Spring BeanDoc.** Ce projet facilite la génération d'une documentation et des diagrammes d'après le fichier de configuration des beans.
- **Spring .NET.** Comme son nom l'indique, ce projet est une version .NET du framework Spring qui facilite le développement d'applications .NET.

2.2 Installer le framework Spring

Problème

Nous souhaitons développer une application Java/Java EE en utilisant le framework Spring. Nous devons commencer par installer ce framework sur notre machine.

Solution

L'installation du framework Spring est très simple. Il suffit de télécharger sa version 2.5 au format ZIP et d'extraire le contenu de l'archive dans un répertoire de notre choix.

Explications

Installer le JDK

Avant d'installer le framework Spring, un JDK doit être présent sur notre machine. Spring 2.5 exige le JDK 1.4 ou une version ultérieure. Il est fortement conseillé d'installer le JDK 1.5 ou une version ultérieure pour utiliser certaines fonctionnalités comme les annotations, le *boxing* et l'*unboxing* automatiques, les fonctions *varargs*, les collections typées (*type-safe*) et les boucles *for-each*.

Installer un IDE Java

Bien que le développement d'applications Java puisse se faire sans IDE, il sera certainement facilité par un tel éditeur. Puisque la configuration de Spring se fait principalement avec du contenu XML, nous devons choisir un IDE qui fournit des fonctions d'édition XML, comme la validation XML et la complétion automatique pour sa syntaxe. Si vous n'avez pas d'IDE Java préféré ou si vous voulez en choisir un autre, nous vous conseillons d'installer Eclipse Web Tools Platform (WTP), disponible en téléchargement à l'adresse **http://www.eclipse.org/webtools/**. Eclipse WTP est très facile à utiliser et propose de nombreuses fonctionnalités puissantes pour Java, le Web et XML. Par ailleurs, il existe un plug-in Spring pour Eclipse qui se nomme Spring IDE.

Télécharger et installer Spring

À partir du site web de Spring, nous pouvons télécharger la version 2.5 du framework. Nous sommes alors redirigés vers SpringSource, où nous devons choisir la distribution de Spring adéquate (voir Figure 2.2).

La Figure 2.2 montre qu'il existe trois distributions de la version 2.5 de Spring. La première contient l'intégralité du framework, y compris les fichiers JAR de Spring, les bibliothèques dépendantes, la documentation, le code source et des exemples d'applications. La deuxième contient uniquement les fichiers JAR de Spring et la documentation.

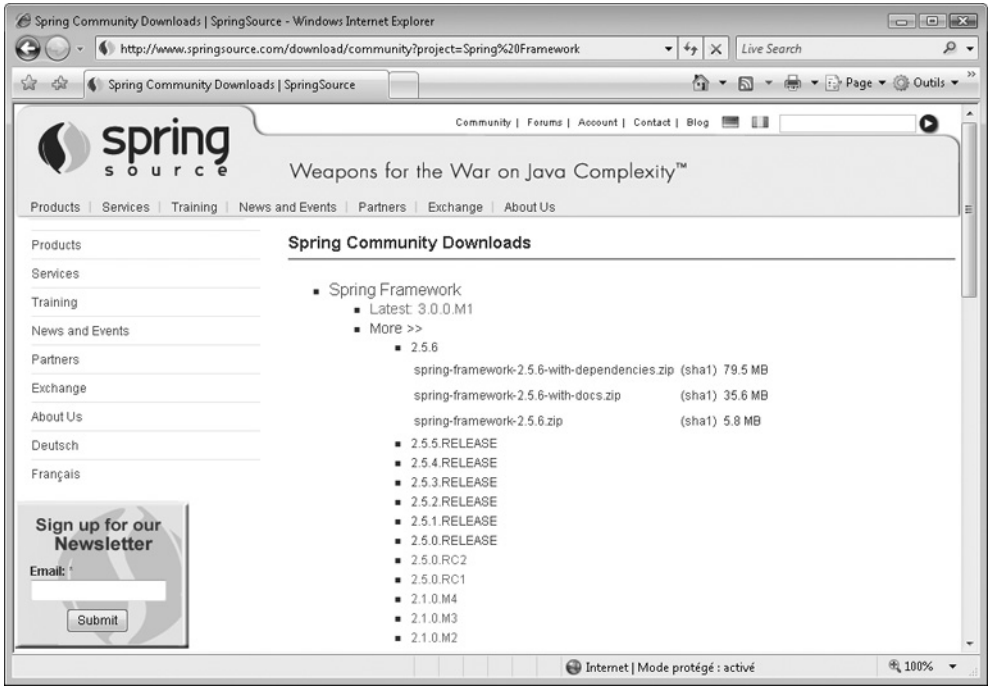


Figure 2.2

Télécharger une distribution de Spring.

La dernière contient uniquement les fichiers JAR de Spring. Il est préférable de télécharger la distribution avec toutes les dépendances, c'est-à-dire `spring-framework-2.5.6-with-dependencies.zip`. Cette archive est la plus volumineuse car elle contient la plupart des bibliothèques dont nous avons besoin pour développer des applications Spring.

Après avoir téléchargé la distribution de Spring, il suffit d'extraire le contenu de l'archive dans le répertoire de notre choix. Cet ouvrage suppose que le développement se fait sur une plate-forme Windows et que Spring est installé dans le répertoire `c:\spring-framework-2.5.6`.

Explorer le répertoire d'installation de Spring

Une fois que la distribution de Spring a été installée, le répertoire de Spring doit contenir les sous-répertoires recensés au Tableau 2.1.

Tableau 2.1 : Sous-répertoires du répertoire d'installation de Spring

Répertoire	*Contenu*
`aspectj`	Le code source et des cas de test pour les aspects Spring utilisés dans AspectJ
`dist`	Les fichiers JAR pour les modules Spring et la prise en charge du tissage, ainsi que les fichiers de ressources, comme les DTD et les XSD
`docs`	La documentation JavaDoc de l'API de Spring, la documentation de référence aux formats PDF et HTML, ainsi qu'un didacticiel pour Spring MVC
`lib`	Les fichiers JAR des bibliothèques utilisées par Spring, organisés par projet
`mock`	Le code source des objets simulacres (*mock object*) de Spring et de la prise en charge des tests
`samples`	Plusieurs exemples d'applications pour illustrer l'utilisation de Spring
`src`	Le code source de la majeure partie du framework Spring
`test`	Les cas de test pour tester le framework Spring
`tiger`	Le code source, les cas de test et les objets simulacres pour Spring spécifiques au JDK 1.5 et les versions ultérieures

2.3 Configurer un projet Spring

Problème

Nous voulons créer un projet Java en utilisant le framework Spring. Nous devons commencer par configurer ce projet.

Solution

Pour configurer un projet Spring, nous devons fixer le chemin d'accès aux classes et créer un ou plusieurs fichiers pour configurer des beans dans le conteneur Spring IoC. Ensuite, nous pouvons instancier le conteneur Spring IoC avec ce fichier de configuration des beans et lui demander des instances de beans.

Explications

Définir le chemin d'accès aux classes

Pour débuter un projet Spring, nous devons inclure les fichiers JAR des modules Spring que nous utilisons et ceux des bibliothèques de dépendance dans le chemin d'accès aux classes. Pour des raisons de commodité, nous pouvons inclure l'unique fichier `spring.jar` qui contient l'ensemble des modules standard, même s'il est relativement

volumineux (environ 2,7 Mo). Il se trouve dans le sous-répertoire `dist` du répertoire d'installation de Spring. Nous pouvons également sélectionner chaque module que nous utilisons à partir du répertoire `dist/modules`.

Un projet Spring a également besoin du fichier `commons-logging.jar`, qui se trouve dans le répertoire `lib/jakarta-commons`. Puisque le framework Spring se sert de cette bibliothèque pour produire ses messages de journalisation, son fichier JAR doit être inclus dans le chemin d'accès aux classes. La configuration de ce chemin varie en fonction des IDE.

Cet ouvrage suppose que les fichiers `spring.jar` et `commons-logging.jar` sont indiqués dans le chemin d'accès aux classes de tous les projets.

Créer le fichier de configuration des beans

Un projet Spring classique nécessite un ou plusieurs fichiers pour configurer les beans dans le conteneur Spring IoC. Nous pouvons placer ces fichiers de configuration dans le chemin d'accès aux classes ou dans un chemin du système de fichiers. Pour faciliter les tests depuis un IDE, il est préférable de le placer dans le chemin d'accès aux classes défini pour le projet. Cet ouvrage suppose que le fichier de configuration des beans est créé à la racine du chemin d'accès aux classes et qu'il se nomme `beans.xml`, sauf précision autre.

Pour illustrer la définition d'un projet Spring, créons la classe `HelloWorld` suivante. Elle accepte une propriété `message` *via* une injection par mutateur. Dans sa méthode `hello()`, nous construisons un message de bienvenue et l'affichons sur la console.

```
package com.apress.springrecipes.hello;

public class HelloWorld {

    private String message;

    public void setMessage(String message) {
        this.message = message;
    }

    public void hello() {
        System.out.println("Bonjour ! " + message);
    }
}
```

Pour configurer un bean de type `HelloWorld` avec un message dans le conteneur Spring IoC, nous créons le fichier suivant. Conformément aux conventions définies pour cet ouvrage, il est placé à la racine du chemin d'accès aux classes et nommé `beans.xml`.

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
```

```
    <bean id="helloWorld" class="com.apress.springrecipes.hello.HelloWorld">
        <property name="message" value="Comment allez-vous ?" />
    </bean>
</beans>
```

Utiliser les beans du conteneur Spring IoC

Nous écrivons la classe `Main` suivante pour instancier le conteneur Spring IoC avec le fichier de configuration des beans nommé `beans.xml` et situé à la racine du chemin d'accès aux classes. Nous pouvons ensuite obtenir le bean `helloWorld` à partir de ce conteneur et l'utiliser comme bon nous semble.

```
package com.apress.springrecipes.hello;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {

    public static void main(String[] args) {
        ApplicationContext context =
            new ClassPathXmlApplicationContext("beans.xml");

        HelloWorld helloWorld = (HelloWorld) context.getBean("helloWorld");
        helloWorld.hello();
    }
}
```

Si tout se passe bien, le message suivant doit apparaître sur la console, après plusieurs lignes de messages de journalisation de Spring :

```
Bonjour ! Comment allez-vous ?
```

2.4 Installer Spring IDE

Problème

Nous souhaitons installer un IDE qui nous aidera lors du développement d'applications Spring.

Solution

Si vous utilisez déjà Eclipse, vous pouvez installer le plug-in Eclipse officiel nommé Spring IDE (**http://springide.org/**). IntelliJ IDEA prend également en charge Spring, mais ses fonctionnalités ne sont pas aussi puissantes ou à jour que Spring IDE. Cet ouvrage se limite à Spring IDE. Il existe trois manières d'installer ce plug-in :

- ajouter le site de mise à jour de Spring IDE (**http://dist.springframework.org/release/IDE**) dans le gestionnaire de mise à jour d'Eclipse et procéder à l'installation en ligne ;

- télécharger le site de mise à jour archivé de Spring IDE et l'installer dans le gestionnaire de mise à jour d'Eclipse en tant que site local ;
- télécharger l'archive de Spring IDE et en extraire le contenu dans le répertoire d'installation d'Eclipse.

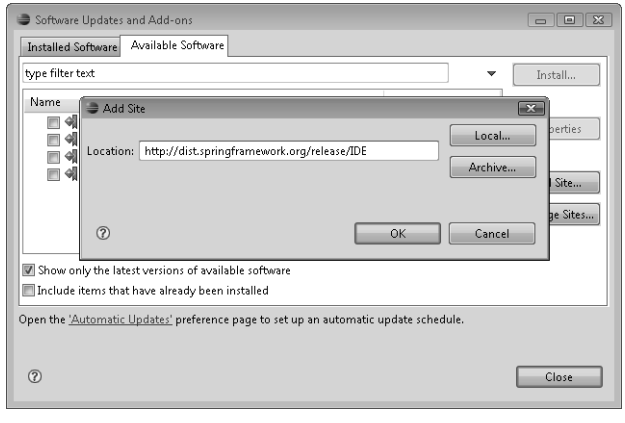
De ces trois méthodes, il est préférable de choisir la première si la machine dispose d'une connexion Internet. Dans le cas contraire, il doit être installé à partir d'un site local archivé. Ces deux solutions permettent de vérifier si tous les plug-in requis par Spring IDE sont déjà installés.

Explications

À partir du menu Help d'Eclipse, choisissez Software Updates. Dans la fenêtre Software Updates and Add-ons qui s'affiche, cliquez sur l'onglet Available Software. Vous voyez alors la liste des sites de mise à jour existants. Si celui correspondant à Spring IDE est absent, ajoutez-le en cliquant sur Add Site. Saisissez alors l'URL du site (voir Figure 2.3).

Figure 2.3

Ajouter le site de mise à jour pour Spring IDE.

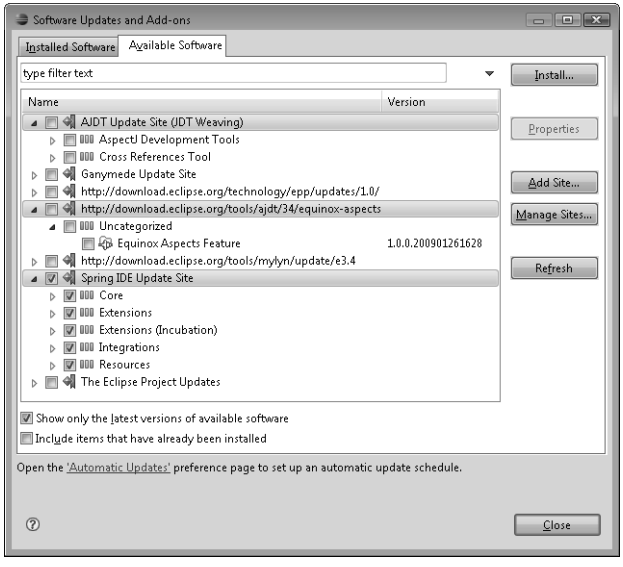


Après avoir cliqué sur OK, la liste des fonctionnalités de Spring IDE et leurs dépendances doivent s'afficher (voir Figure 2.4).

Spring IDE 2.2.1 utilise deux autres plug-in Eclipse : Eclipse Mylyn et AJDT (*AspectJ Development Tools*). Mylyn est une interface utilisateur orientée tâche qui peut intégrer de multiples tâches de développement et diminuer la surcharge d'informations. AJDT est un plug-in Eclipse pour le développement AspectJ. Spring IDE se fonde sur AJDT pour la prise en charge du développement orienté aspect de Spring 2.0. Vous pouvez également installer les fonctionnalités d'intégration de Spring IDE avec ces plug-in.

Figure 2.4

Sélectionner les fonctionnalités de Spring IDE qui seront installées.



Si vous utilisez Eclipse 3.3 ou une version ultérieure, vous ne devez pas sélectionner la catégorie Dependencies dans Spring IDE car elle est réservée à Eclipse 3.2. Pour installer Spring IDE, il suffit ensuite de suivre les instructions pas à pas.

2.5 Utiliser les outils de gestion des beans de Spring IDE

Problème

Nous voulons employer les fonctionnalités de prise en charge des beans de Spring IDE pour nous aider à développer des applications Spring.

Solution

Spring IDE propose de riches fonctions de gestion des beans qui peuvent améliorer notre productivité lors du développement d'applications Spring. En utilisant Spring IDE, nous pouvons afficher nos beans en mode explorateur ou graphique. Par ailleurs, Spring IDE peut nous assister dans l'écriture du contenu du fichier de configuration des beans et vérifier sa conformité.

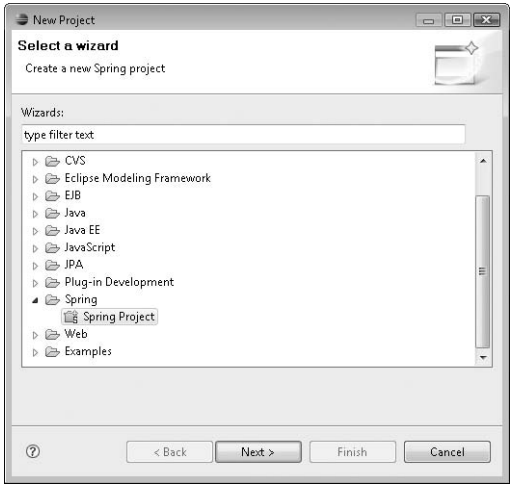
Explications

Créer un projet Spring

Pour créer un projet Spring dans Eclipse lorsque Spring IDE est installé, il suffit de choisir File > New > Project et de sélectionner Spring Project dans la rubrique Spring (voir Figure 2.5).

Figure 2.5

Créer un projet Spring.



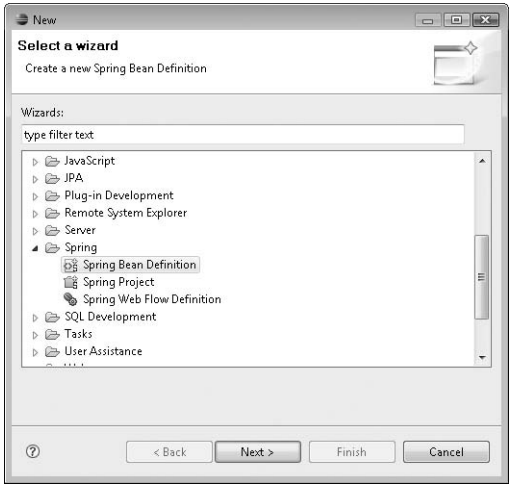
Pour convertir un projet Java existant en projet Spring, nous sélectionnons ce projet, cliquons dessus du bouton droit et choisissons Spring Tools > Add Spring Project Nature pour activer la prise en charge de Spring IDE sur ce projet. L'icône d'un projet Spring comprend un petit "S" dans le coin supérieur droit.

Créer un fichier de configuration des beans

Pour créer un fichier de configuration des beans, nous choisissons File > New > Other et sélectionnons Spring Bean Definition dans la rubrique Spring (voir Figure 2.6).

Figure 2.6

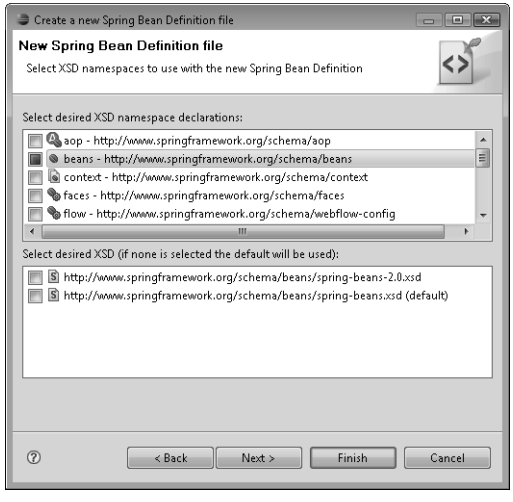
Créer un fichier de configuration des beans Spring.



Après avoir choisi l'emplacement et le nom de ce fichier de configuration des beans, nous devons indiquer les espaces de noms XSD à inclure dans ce fichier (voir Figure 2.7).

Figure 2.7

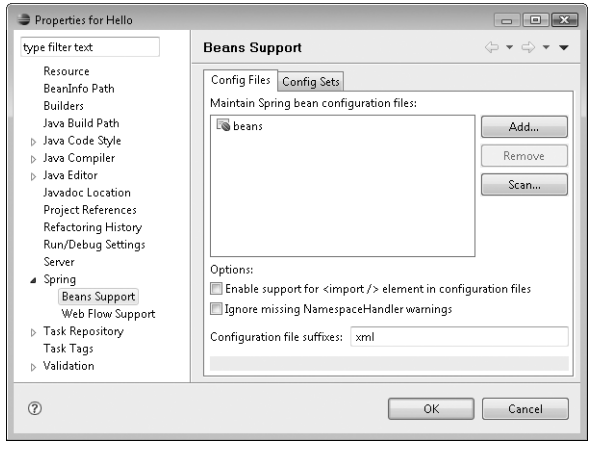
Sélectionner les espaces de noms XSD à inclure dans le fichier de configuration des beans.



Pour utiliser un fichier XML existant comme fichier de configuration des beans, il suffit de cliquer du bouton droit sur le projet et de choisir Properties. Depuis Beans Support, qui se trouve dans la rubrique Spring, nous pouvons ajouter nos fichiers XML (voir Figure 2.8).

Figure 2.8

Indiquer les fichiers de configuration des beans dans un projet Spring.



Visualiser les beans dans Spring Explorer

Pour mieux illustrer les fonctions d'exploration et les fonctions graphiques de Spring IDE, créons la classe `Holiday` suivante. Pour des questions de simplicité, les accesseurs et les mutateurs ne sont pas présentés. La commande Source > Generate Getters and Setters permet de les générer facilement.

```
package com.apress.springrecipes.hello;

public class Holiday {

    private int month;
    private int day;
    private String greeting;

    // Accesseurs et mutateurs.
    ...
}
```

Nous modifions ensuite notre classe `HelloWorld` pour lui ajouter la propriété `holidays` de type `java.util.List` et le mutateur correspondant.

```
package com.apress.springrecipes.hello;

import java.util.List;

public class HelloWorld {
    ...
    private List<Holiday> holidays;

    public void setHolidays(List<Holiday> holidays) {
        this.holidays = holidays;
    }
}
```

Dans le fichier de configuration des beans, nous déclarons les beans `christmas` et `newYear` de type `Holiday`. Ensuite, nous ajoutons des références à ces deux beans dans la propriété `holidays` du bean `helloWorld`.

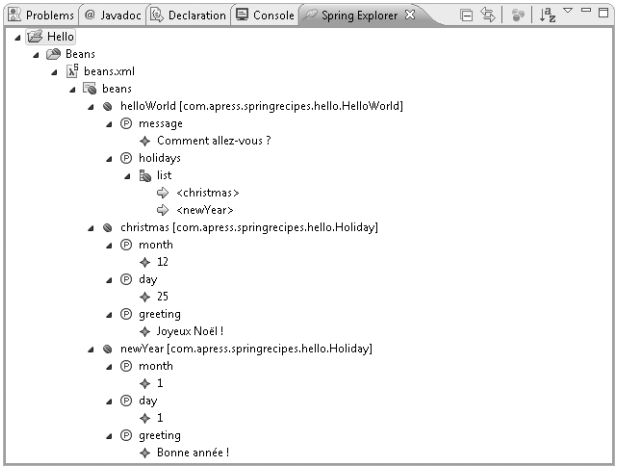
```
<beans ...>
    <bean id="helloWorld" class="com.apress.springrecipes.hello.HelloWorld">
        <property name="message" value="Comment allez-vous ?" />
        <property name="holidays">
            <list>
                <ref local="christmas" />
                <ref local="newYear" />
            </list>
        </property>
    </bean>

    <bean id="christmas" class="com.apress.springrecipes.hello.Holiday">
        <property name="month" value="12" />
        <property name="day" value="25" />
        <property name="greeting" value="Joyeux Noël !" />
    </bean>
```

```
    <bean id="newYear" class="com.apress.springrecipes.hello.Holiday">
        <property name="month" value="1" />
        <property name="day" value="1" />
        <property name="greeting" value="Bonne année !" />
    </bean>
</beans>
```

Pour visualiser nos beans dans Spring Explorer, nous cliquons du bouton droit sur le fichier de configuration des beans et choisissons Show In > Spring Explorer (voir Figure 2.9).

Figure 2.9

Explorer les beans Spring dans Spring Explorer.



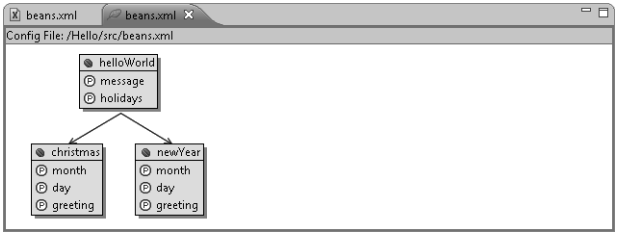
Dans Spring Explorer, nous pouvons double-cliquer sur un élément pour aller directement à sa déclaration dans le fichier de configuration des beans. Il est ainsi très commode de modifier cet élément.

Visualiser des beans Spring dans Spring Beans Graph

Nous pouvons cliquer du bouton droit sur un élément de bean (ou l'élément racine) dans Spring Explorer et choisir Open Graph pour visualiser les beans sous forme de graphe (voir Figure 2.10).

Figure 2.10

Visualiser les beans Spring dans Spring Beans Graph.



Utiliser l'assistance de contenu dans les fichiers de configuration des beans

Spring IDE prend en charge la fonctionnalité d'assistance de contenu d'Eclipse pour modifier les noms de classes, les noms de propriétés et les références aux noms de beans. Par défaut, la combinaison de touches qui active cette fonctionnalité est Ctrl+Barre d'espace. Nous pouvons par exemple l'employer pour compléter le nom d'une propriété (voir Figure 2.11).

Figure 2.11

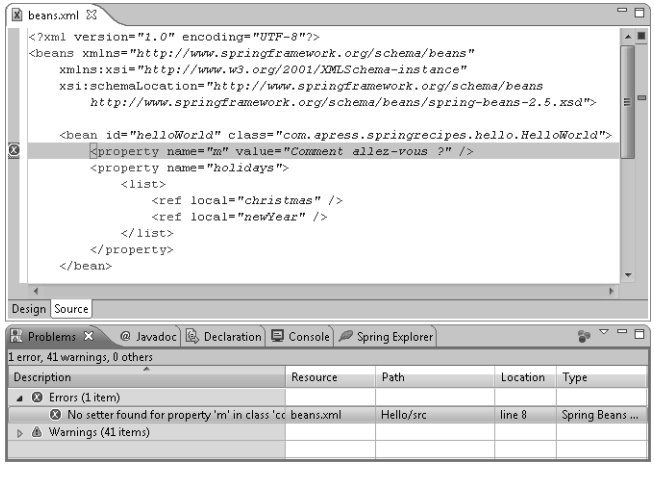
Utiliser l'assistance de contenu dans les fichiers de configuration des beans.

Valider les fichiers de configuration des beans

Lors de l'enregistrement du fichier de configuration des beans, Spring IDE vérifie automatiquement sa conformité par rapport aux classes des beans, aux noms des propriétés, aux références aux noms de beans, etc. Toute erreur est signalée (voir Figure 2.12).

Figure 2.12

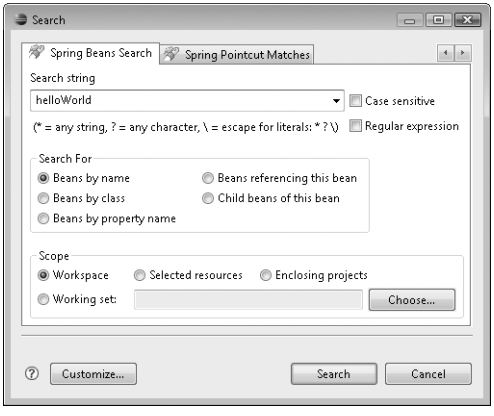
Valider les fichiers de configuration des beans avec Spring IDE.



Rechercher des beans Spring

Spring IDE prend en charge la recherche de beans répondant à certains critères. Dans le menu Search, choisissez Beans pour afficher la boîte de dialogue de recherche de beans (voir Figure 2.13).

Figure 2.13
Rechercher des beans Spring.



2.6 En résumé

Dans ce chapitre, nous avons examiné l'architecture globale et les principales fonctions du framework Spring, ainsi que l'objectif général de chaque sous-projet. L'installation de Spring est très simple. Il suffit de le télécharger et d'extraire son contenu dans un répertoire. Nous avons eu un aperçu du contenu de chaque sous-répertoire du répertoire d'installation de Spring. Nous avons créé un projet Spring très simple, qui a illustré la configuration générale des projets. Nous en avons également profité pour définir les conventions qui seront employées tout au long de cet ouvrage.

L'équipe de développement de Spring a créé un plug-in Eclipse, nommé Spring IDE, pour simplifier le développement d'applications Spring. Dans ce chapitre, nous avons installé cet IDE et appris à utiliser ses fonctionnalités de base pour la gestion des beans.

Le chapitre suivant présente les bases de la configuration d'un bean dans Spring IoC.

3

Configuration des beans

Au sommaire de ce chapitre

- Configurer des beans dans le conteneur Spring IoC
- Instancier le conteneur Spring IoC
- Lever les ambiguïtés sur le constructeur
- Préciser des références de beans
- Contrôler les propriétés par vérification des dépendances
- Contrôler les propriétés avec l'annotation *@Required*
- Lier automatiquement des beans par configuration XML
- Lier automatiquement des beans avec *@Autowired* et *@Resource*
- Hériter de la configuration d'un bean
- Affecter des collections aux propriétés de bean
- Préciser le type de données des éléments d'une collection
- Définir des collections avec des beans de fabrique et le schéma *util*
- Rechercher les composants dans le chemin d'accès aux classes
- En résumé

Ce chapitre détaille les bases de la configuration d'un composant dans le conteneur Spring IoC. Placé au cœur du framework Spring, ce conteneur est conçu pour être hautement configurable et bénéficier d'une grande capacité d'adaptation. Ses fonctionnalités simplifient autant que possible la configuration des composants qui devront s'y exécuter.

Dans Spring, les composants sont également appelés "beans". Notez que ce concept est différent de celui défini par Sun dans la spécification JavaBeans. Rien n'oblige à ce que les beans déclarés dans le conteneur Spring IoC soient des JavaBeans. Ils peuvent être des objets Java tout simples (POJO, *Plain Old Java Object*). Le terme *POJO* fait référence à un objet Java ordinaire qui ne répond à aucune exigence particulière, comme

implémenter une interface spécifique ou dériver d'une certaine classe de base. Il permet de distinguer les composants Java légers des composants lourds existant dans d'autres modèles de composants complexes (par exemple les composants EJB dans les versions antérieures à la 3.0).

À la fin de ce chapitre, vous serez capable de construire une application Java complète fondée sur le conteneur Spring IoC. Par ailleurs, si vous examinez alors vos anciennes applications Java, vous pourriez constater qu'elles pourraient être grandement simplifiées et améliorées en utilisant ce conteneur.

3.1 Configurer des beans dans le conteneur Spring IoC

Problème

Spring fournit un conteneur IoC puissant pour la gestion des beans qui composent une application. Pour bénéficier des services de ce conteneur, nous devons configurer nos beans de telle sorte qu'ils s'y exécutent.

Solution

La configuration des beans dans le conteneur Spring IoC peut se faire avec des fichiers XML, des fichiers de propriétés ou même des API. En raison de sa simplicité et de sa maturité, la configuration à base de fichiers XML a été choisie dans cet ouvrage. Si vous êtes intéressé par les autres méthodes, consultez la documentation de Spring, qui détaille tous les aspects de la configuration des beans.

Spring permet de configurer les beans dans un ou plusieurs fichiers de configuration. Dans le cas d'une application simple, nous pouvons regrouper les beans dans un seul fichier. En revanche, pour une application plus complexe, avec un grand nombre de beans, il est préférable de les répartir dans plusieurs fichiers de configuration selon leurs fonctionnalités.

Explications

Supposons que nous développions une application pour la génération séquentielle de numéros. Elle doit pouvoir contenir différents ensembles de numéros en séquence pour répondre à des objectifs variés. Chaque ensemble dispose de ses propres valeurs de préfixe, de suffixe et de départ. Nous devons par conséquent créer et maintenir plusieurs instances du générateur dans notre application.

Créer la classe du bean

Conformément aux exigences, nous créons la classe `SequenceGenerator` avec les trois propriétés `prefix`, `suffix` et `initial`, qui peuvent être injectées par l'intermédiaire de mutateurs ou du constructeur. Le champ privé `counter` enregistre la valeur numérique actuelle du générateur. Chaque fois que nous invoquons la méthode `getSequence()` sur une instance du générateur, nous recevons le dernier numéro de séquence, auquel sont ajoutés le préfixe et le suffixe. Nous déclarons cette méthode `synchronized` pour qu'elle soit sûre vis-à-vis des threads.

```
package com.apress.springrecipes.sequence;

public class SequenceGenerator {

    private String prefix;
    private String suffix;
    private int initial;
    private int counter;

    public SequenceGenerator() {}

    public SequenceGenerator(String prefix, String suffix, int initial) {
        this.prefix = prefix;
        this.suffix = suffix;
        this.initial = initial;
    }

    public void setPrefix(String prefix) {
        this.prefix = prefix;
    }

    public void setSuffix(String suffix) {
        this.suffix = suffix;
    }

    public void setInitial(int initial) {
        this.initial = initial;
    }

    public synchronized String getSequence() {
        StringBuffer buffer = new StringBuffer();
        buffer.append(prefix);
        buffer.append(initial + counter++);
        buffer.append(suffix);
        return buffer.toString();
    }
}
```

Cette classe `SequenceGenerator` applique à la fois l'injection par constructeur et l'injection par mutateur pour les propriétés `prefix`, `suffix` et `initial`.

Créer le fichier de configuration des beans

Pour déclarer des beans dans le conteneur Spring IoC, nous devons tout d'abord créer un fichier XML de configuration des beans et le nommer de manière appropriée, par exemple `beans.xml`. Ce fichier est placé à la racine du chemin d'accès aux classes pour faciliter les tests depuis un IDE. Au début du fichier, nous indiquons la DTD Spring 2.0 de manière à importer la structure valide d'un fichier de configuration des beans pour Spring 2.x. Il est possible de définir un ou plusieurs beans sous l'élément racine `<beans>`.

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
    "http://www.springframework.org/dtd/spring-beans-2.0.dtd">

<beans>
    ...
</beans>
```

Spring affiche une rétrocompatibilité au niveau de sa configuration. Nous pouvons donc utiliser nos fichiers de configuration 1.0 existants (avec la DTD Spring 1.0) dans Spring 2.x. Toutefois, en procédant ainsi, nous ne pourrons pas bénéficier des nouvelles fonctions de configuration ajoutées dans Spring 2.x. L'usage des anciens fichiers de configuration doit être réservé à une phase de transition.

Spring 2.x accepte également d'utiliser XSD pour définir la structure valide du fichier XML de configuration des beans. XSD présente de nombreux avantages par rapport à une DTD classique. Dans Spring 2.x, le plus important est qu'il nous permet d'utiliser des balises personnalisées issues de différents schémas pour simplifier et clarifier la configuration. Il est donc fortement recommandé de choisir Spring XSD à la place de la DTD dès lors que c'est possible. Spring XSD est une version spécifique mais rétrocompatible. Si nous utilisons Spring 2.5, nous devons choisir Spring 2.5 XSD pour bénéficier des nouveautés de la version 2.5.

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
    ...
</beans>
```

Déclarer des beans dans le fichier de configuration

Pour que le conteneur Spring IoC puisse l'instancier, chaque bean doit posséder un nom unique et un nom de classe complet. Pour chaque propriété de type simple, par exemple `String` et les autres types primitifs, nous pouvons ajouter un élément `<value>`. Spring se charge de convertir la valeur dans le type déclaré de cette propriété. Pour configurer une propriété à l'aide de l'injection par mutateur, nous utilisons l'élément `<property>` et indiquons le nom de la propriété dans l'attribut `name`.

```
<bean name="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator">
    <property name="prefix">
        <value>30</value>
    </property>
    <property name="suffix">
        <value>A</value>
    </property>
    <property name="initial">
        <value>100000</value>
    </property>
</bean>
```

Nous pouvons également configurer les propriétés du bean à l'aide de l'injection par constructeur. Pour cela, il suffit de les déclarer dans des éléments `<constructor-arg>`. Cet élément ne gère pas d'attribut `name` car les arguments du constructeur sont définis par leur emplacement.

```
<bean name="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator">
    <constructor-arg>
        <value>30</value>
    </constructor-arg>
    <constructor-arg>
        <value>A</value>
    </constructor-arg>
    <constructor-arg>
        <value>100000</value>
    </constructor-arg>
</bean>
```

Dans le conteneur Spring IoC, le nom de chaque bean doit être unique, même s'il est possible d'employer plusieurs fois le même nom pour modifier une déclaration. Le nom d'un bean peut être donné dans l'attribut `name` de l'élément `<bean>`. Cependant, il existe une autre solution, recommandée. Elle consiste à utiliser l'attribut `id` standard, qui identifie un élément dans un document XML. Si l'éditeur de texte comprend la syntaxe XML, il peut ainsi nous aider à vérifier l'unicité de chaque bean lors de la conception.

```
<bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator">
    ...
</bean>
```

XML impose certaines restrictions sur les caractères acceptés dans l'attribut `id`, mais, en général, personne n'utilise de caractères spéciaux dans un nom de bean. Par ailleurs, avec l'attribut `name`, il est possible de donner plusieurs noms à un bean, en les séparant par des virgules. Ce n'est pas le cas avec l'attribut `id` car les virgules n'y sont pas acceptées.

En réalité, rien ne nous oblige à nommer ou à identifier le bean. Il s'agit alors d'un *bean anonyme*.

Définir les propriétés du bean avec des raccourcis

Spring propose un raccourci pour préciser la valeur d'une propriété de type simple : définir l'attribut `value` dans l'élément `<property>` au lieu d'ajouter un sous-élément `<value>`.

```
<bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator">
    <property name="prefix" value="30" />
    <property name="suffix" value="A" />
    <property name="initial" value="100000" />
</bean>
```

Ce raccourci fonctionne également avec les arguments du constructeur.

```
<bean name="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator">
    <constructor-arg value="30" />
    <constructor-arg value="A" />
    <constructor-arg value="100000" />
</bean>
```

Spring 2.x propose un autre raccourci pratique pour définir des propriétés. Il s'agit d'utiliser le schéma `p` et de définir les propriétés sous forme d'attributs dans l'élément `<bean>`. Les lignes de la configuration XML sont ainsi plus courtes.

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <bean id="sequenceGenerator"
        class="com.apress.springrecipes.sequence.SequenceGenerator"
        p:prefix="30" p:suffix="A" p:initial="100000" />
</beans>
```

3.2 Instancier le conteneur Spring IoC

Problème

Nous devons instancier le conteneur Spring IoC pour qu'il crée des instances de beans d'après leur configuration. Ensuite, nous voulons obtenir des instances de beans à partir du conteneur pour les utiliser.

Solution

Spring fournit deux implémentations du conteneur IoC. La version basique est appelée *fabrique de beans*. La version plus élaborée est appelée *contexte d'application* ; elle est compatible avec la fabrique de beans. Ces deux types de conteneurs IoC utilisent des fichiers de configuration des beans identiques.

Le contexte d'application offre des fonctions plus élaborées que la fabrique de beans, tout en conservant la compatibilité des fonctionnalités de base. Par conséquent, il est fortement conseillé d'utiliser le contexte d'application, excepté lorsque les ressources de l'application sont limitées, par exemple lorsqu'elle s'exécute dans une applet ou sur un périphérique mobile.

Les interfaces de la fabrique de beans et du contexte d'application sont respectivement `BeanFactory` et `ApplicationContext`. Cette dernière étend `BeanFactory` pour assurer la compatibilité.

Explications

Instancier une fabrique de beans

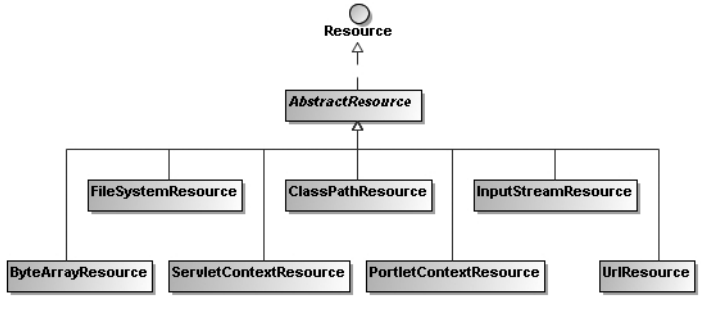
Pour instancier une fabrique de beans, nous devons commencer par charger le fichier de configuration des beans dans un objet `Resource`. Par exemple, l'instruction suivante charge le fichier de configuration à partir de la racine du chemin d'accès aux classes :

```
Resource resource = new ClassPathResource("beans.xml");
```

`Resource` n'est qu'une interface, tandis que `ClassPathResource` est l'une de ses implémentations pour charger une ressource à partir du chemin d'accès aux classes. D'autres implémentations de cette interface, comme `FileSystemResource`, `InputStreamResource` et `UrlResource`, permettent de charger une ressource à partir d'emplacements différents. La Figure 3.1 présente les implémentations de l'interface `Resource` proposées par Spring.

Figure 3.1

Implémentations communes de l'interface `Resource`.



Ensuite, nous utilisons l'instruction suivante pour instancier une fabrique de beans en lui passant l'objet `Resource` dans lequel a été chargé le fichier de configuration :

```
BeanFactory factory = new XmlBeanFactory(resource);
```

Nous l'avons mentionné, `BeanFactory` est une interface qui permet de rendre abstraites les opérations sur une fabrique de beans, tandis que `XmlBeanFactory` est l'implémentation qui construit une fabrique de beans à partir d'un fichier de configuration XML.

Instancier un contexte d'application

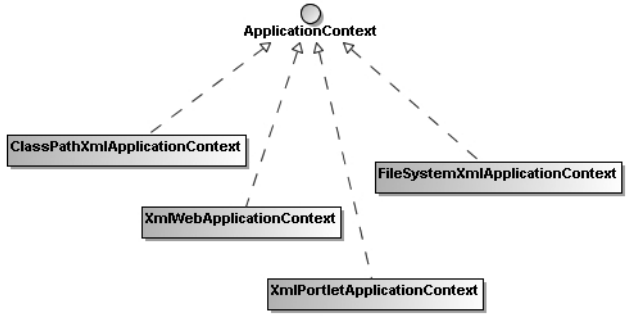
Comme `BeanFactory`, `ApplicationContext` est une interface et nous devons donc instancier l'une de ses implémentations. La classe `ClassPathXmlApplicationContext` construit un contexte d'application en chargeant un fichier de configuration XML à partir du chemin d'accès aux classes. Nous pouvons également lui indiquer plusieurs fichiers de configuration.

```
ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");
```

Outre `ClassPathXmlApplicationContext`, Spring propose d'autres implémentations d'`ApplicationContext`. `FileSystemXmlApplicationContext` charge des fichiers de configuration XML à partir du système de fichiers, tandis que `XmlWebApplicationContext` et `XmlPortletApplicationContext` peuvent uniquement être utilisées dans des applications et des portails web. La Figure 3.2 présente les implémentations de l'interface `ApplicationContext` disponibles dans Spring.

Figure 3.2

Implémentations communes de l'interface `ApplicationContext`.



Obtenir des beans depuis le conteneur IoC

Pour obtenir un bean déclaré à partir d'une fabrique de beans ou d'un contexte d'application, il suffit d'invoquer la méthode `getBean()` en lui passant le nom unique du bean. Cette méthode retourne un `java.lang.Object`, dont nous devons forcer le type avant de pouvoir l'utiliser.

```
SequenceGenerator generator =
    (SequenceGenerator) context.getBean("sequenceGenerator");
```

À partir de là, nous pouvons employer le bean comme n'importe quel autre objet créé à l'aide d'un constructeur. L'intégralité du code source pour l'exécution du générateur séquentiel de numéros est donnée dans la classe `Main` suivante :

```
package com.apress.springrecipes.sequence;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {

    public static void main(String[] args) {
        ApplicationContext context =
            new ClassPathXmlApplicationContext("beans.xml");

        SequenceGenerator generator =
            (SequenceGenerator) context.getBean("sequenceGenerator");

        System.out.println(generator.getSequence());
        System.out.println(generator.getSequence());
    }
}
```

Si tout se passe bien, nous voyons s'afficher les numéros de séquence suivants, après quelques messages de journalisation qui ne nous intéressent pas :

```
30100000A
30100001A
```

3.3 Lever les ambiguïtés sur le constructeur

Problème

Lorsque nous indiquons un ou plusieurs arguments de constructeur pour un bean, Spring tente de trouver le constructeur approprié dans la classe du bean et lui passe nos arguments lors de l'instanciation. Cependant, si la liste des arguments correspond à plusieurs constructeurs, il existe une ambiguïté sur le choix du constructeur. Dans ce cas, Spring pourrait ne pas invoquer le constructeur attendu.

Solution

Nous pouvons renseigner les attributs `type` et `index` de l'élément `<constructor-arg>` de manière à aider Spring dans sa recherche du constructeur.

Explications

Ajoutons un nouveau constructeur à la classe `SequenceGenerator` avec les arguments `prefix` et `suffix`.

```
package com.apress.springrecipes.sequence;

public class SequenceGenerator {
    ...
    public SequenceGenerator(String prefix, String suffix) {
        this.prefix = prefix;
        this.suffix = suffix;
    }
}
```

Dans la déclaration du bean, nous pouvons préciser un ou plusieurs arguments du constructeur grâce à des éléments `<constructor-arg>`. Spring essaiera de trouver le constructeur pour cette classe et lui passera nos arguments lors de l'instanciation du bean. Il ne faut pas oublier que l'attribut `name` n'existe pas dans l'élément `<constructor-arg>`, car les arguments du constructeur sont fonction de leur emplacement.

```
<bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator">
    <constructor-arg value="30" />
    <constructor-arg value="A" />
    <property name="initial" value="100000" />
</bean>
```

Spring n'a aucun mal à trouver un constructeur correspondant à ces deux arguments car il n'en existe qu'un seul. Supposons que nous ayons ajouté un autre constructeur à `SequenceGenerator`, avec les arguments `prefix` et `initial`.

```
package com.apress.springrecipes.sequence;

public class SequenceGenerator {
    ...
    public SequenceGenerator(String prefix, String suffix) {
        this.prefix = prefix;
        this.suffix = suffix;
    }

    public SequenceGenerator(String prefix, int initial) {
        this.prefix = prefix;
        this.initial = initial;
    }
}
```

Pour invoquer ce constructeur en passant un préfixe et une valeur initiale, nous déclarons le bean de la manière suivante. Le suffixe est injecté *via* le mutateur.

```
<bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator">
    <constructor-arg value="30" />
    <constructor-arg value="100000" />
    <property name="suffix" value="A" />
</bean>
```

Cependant, si nous exécutons à présent l'application, nous obtenons le résultat suivant :

```
300A
301A
```

Ce résultat inattendu provient de l'invocation du premier constructeur, avec `prefix` et `suffix` comme arguments, au lieu du second. En effet, Spring a converti par défaut nos deux arguments en type `String` et a considéré que le premier constructeur était mieux adapté car il n'exigeait aucune conversion de type. Pour préciser le type attendu de nos arguments, nous utilisons l'attribut `type` de `<constructor-arg>`.

```
<bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator">
    <constructor-arg type="java.lang.String" value="30" />
    <constructor-arg type="int" value="100000" />
    <property name="suffix" value="A" />
</bean>
```

Ajoutons un autre constructeur à `SequenceGenerator`, avec `initial` et `suffix` comme arguments, et modifions en conséquence la déclaration du bean.

```
package com.apress.springrecipes.sequence;

public class SequenceGenerator {
    ...
    public SequenceGenerator(String prefix, String suffix) {
        this.prefix = prefix;
        this.suffix = suffix;
    }

    public SequenceGenerator(String prefix, int initial) {
        this.prefix = prefix;
        this.initial = initial;
    }

    public SequenceGenerator(int initial, String suffix) {
        this.initial = initial;
        this.suffix = suffix;
    }
}

---

<bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator">
    <constructor-arg type="int" value="100000" />
    <constructor-arg type="java.lang.String" value="A" />
    <property name="prefix" value="30" />
</bean>
```

Si nous exécutons à nouveau l'application, nous pouvons obtenir le bon résultat ou le résultat inattendu suivant :

```
30100000null
30100001null
```

Cette incertitude vient du fait que Spring évalue de manière interne la compatibilité de chaque constructeur avec nos arguments. Au cours du processus d'évaluation, l'ordre d'apparition de nos arguments dans le fichier XML n'est pas pris en compte. Autrement

dit, du point de vue de Spring, le deuxième et le troisième constructeurs sont équivalents. Le constucteur choisi est celui trouvé en premier. Selon l'API Java Reflection ou, plus précisément, la méthode `Class.getDeclaredConstructors()`, les constructeurs retournés sont dans un ordre quelconque qui peut varier de l'ordre de déclaration. En raison de tous ces éléments, il existe une ambiguïté sur la correspondance du constructeur.

Pour éviter ce problème, nous devons indiquer explicitement les indices des arguments à l'aide de l'attribut `index` de `<constructor-arg>`. Lorsque les attributs `type` et `index` sont utilisés, Spring est en mesure de trouver précisément le constructeur voulu d'un bean.

```
<bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator">
    <constructor-arg type="int" index="0" value="100000" />
    <constructor-arg type="java.lang.String" index="1" value="A" />
    <property name="prefix" value="30" />
</bean>
```

Si nous sommes certains que nos constructeurs ne seront pas source d'ambiguïté, nous pouvons omettre les attributs `type` et `index`.

3.4 Préciser des références de beans

Problème

Les beans qui composent une application doivent souvent collaborer les uns avec les autres pour mettre en œuvre ses fonctionnalités. Pour cela, nous devons préciser les différentes références entre les beans dans le fichier de configuration des beans.

Solution

Dans le fichier de configuration, nous pouvons affecter une référence de bean à une propriété de bean ou à un argument de constructeur à l'aide de l'élément `<ref>`. La procédure n'est pas plus compliquée que la définition d'une simple valeur avec l'élément `<value>`. Nous pouvons également placer directement une déclaration de bean dans une propriété ou un argument de constructeur sous forme de bean interne.

Explications

Pour le moment, la seule valeur acceptée comme préfixe pour notre générateur séquentiel est une chaîne de caractères. Cela n'est évidemment pas suffisamment souple pour répondre aux exigences futures. Il serait préférable que le préfixe soit fourni par une forme de logique programmée. Nous créons donc l'interface `PrefixGenerator` pour définir l'opération de génération du préfixe.

```
package com.apress.springrecipes.sequence;

public interface PrefixGenerator {

    public String getPrefix();
}
```

Une stratégie de génération consiste à mettre en forme la date système actuelle à l'aide d'un motif spécifique. Nous créons la classe `DatePrefixGenerator`, qui implémente l'interface `PrefixGenerator`.

```
package com.apress.springrecipes.sequence;
...
public class DatePrefixGenerator implements PrefixGenerator {

    private DateFormat formatter;

    public void setPattern(String pattern) {
        this.formatter = new SimpleDateFormat(pattern);
    }

    public String getPrefix() {
        return formatter.format(new Date());
    }
}
```

Le motif est injecté par l'intermédiaire du mutateur `setPattern()` et sert ensuite à créer un objet `java.text.DateFormat` de mise en forme de la date. Puisque la chaîne du motif n'est plus utilisée après avoir créé l'objet `DateFormat`, il est inutile de l'enregistrer dans un champ privé.

Nous pouvons à présent déclarer un bean de type `DatePrefixGenerator` avec une chaîne de motif quelconque pour mettre en forme la date.

```
<bean id="datePrefixGenerator"
    class="com.apress.springrecipes.sequence.DatePrefixGenerator">
    <property name="pattern" value="yyyyMMdd" />
</bean>
```

Préciser les références de beans pour les mutateurs

Pour profiter de cette génération du préfixe, la classe `SequenceGenerator` doit accepter un objet de type `PrefixGenerator` à la place d'une simple chaîne de caractères. Pour cela, nous pouvons choisir l'injection par mutateur. Nous supprimons la propriété `prefix`, ainsi que ses mutateurs et constructeurs, qui sont source d'erreurs de compilation.

```
package com.apress.springrecipes.sequence;

public class SequenceGenerator {
    ...
    private PrefixGenerator prefixGenerator;
```

```
    public void setPrefixGenerator(PrefixGenerator prefixGenerator) {
        this.prefixGenerator = prefixGenerator;
    }

    public synchronized String getSequence() {
        StringBuffer buffer = new StringBuffer();
        buffer.append(prefixGenerator.getPrefix());
        buffer.append(initial + counter++);
        buffer.append(suffix);
        return buffer.toString();
    }
}
```

Ensuite, un bean `SequenceGenerator` peut faire référence au bean `datePrefixGenerator` dans sa propriété `prefixGenerator` en utilisant un élément `<ref>`.

```
<bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator">
    <property name="initial" value="100000" />
    <property name="suffix" value="A" />
    <property name="prefixGenerator">
        <ref bean="datePrefixGenerator" />
    </property>
</bean>
```

Le nom du bean dans l'attribut `bean` de l'élément `<ref>` peut être une référence à n'importe quel bean du conteneur IoC, même s'il est défini dans un autre fichier de configuration XML. Dans le cas d'une référence à un bean présent dans le même fichier XML, nous pouvons employer l'attribut `local` puisqu'il s'agit d'une référence à un identifiant XML. Notre éditeur XML peut alors vérifier si un bean possédant cet identifiant existe bien dans le même fichier XML (intégrité de référence).

```
<bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator">
    ...
    <property name="prefixGenerator">
        <ref local="datePrefixGenerator" />
    </property>
</bean>
```

Il existe également un raccourci qui permet de préciser une référence de bean dans l'attribut `ref` d'un élément `<property>`.

```
<bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator">
    ...
    <property name="prefixGenerator" ref="datePrefixGenerator" />
</bean>
```

Toutefois, cette méthode ne permet pas à notre éditeur XML de vérifier l'intégrité des références. En réalité, elle équivaut à utiliser l'attribut `bean` dans un élément `<ref>`.

Spring 2.x propose un autre raccourci pour préciser des références de beans : utiliser le schéma `p` et préciser les références sous forme d'attributs dans l'élément `<bean>`. Les lignes de la configuration XML sont ainsi plus courtes.

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <bean id="sequenceGenerator"
        class="com.apress.springrecipes.sequence.SequenceGenerator"
        p:suffix="A" p:initial="1000000"
        p:prefixGenerator-ref="datePrefixGenerator" />
</beans>
```

Pour différencier une référence de bean d'une simple valeur de propriété, nous devons ajouter le suffixe `-ref` au nom de la propriété.

Préciser des références de beans pour les arguments d'un constructeur

Des références de beans peuvent également être appliquées en utilisant l'injection par constructeur. Ajoutons, par exemple, un constructeur qui accepte un objet `PrefixGenerator` en argument.

```
package com.apress.springrecipes.sequence;

public class SequenceGenerator {
    ...
    private PrefixGenerator prefixGenerator;

    public SequenceGenerator(PrefixGenerator prefixGenerator) {
        this.prefixGenerator = prefixGenerator;
    }
}
```

Comme nous l'avons fait dans l'élément `<property>`, nous incluons une référence de bean avec `<ref>` dans l'élément `<constructor-arg>`.

```
<bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator">
    <constructor-arg>
        <ref local="datePrefixGenerator" />
    </constructor-arg>
    <property name="initial" value="100000" />
    <property name="suffix" value="A" />
</bean>
```

Le raccourci précédent pour indiquer une référence de bean fonctionne également avec `<constructor-arg>`.

```
<bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator">
    <constructor-arg ref="datePrefixGenerator" />
    ...
</bean>
```

Déclarer des beans internes

Dès lors qu'une instance de bean n'est utilisée que dans une seule propriété, elle peut être déclarée sous forme de bean interne. Une déclaration de bean interne est incluse directement dans un élément `<property>` ou `<constructor-arg>`, sans préciser d'attribut `id` ou `name`. De cette manière, le bean est anonyme et ne peut pas être employé ailleurs. En réalité, si nous affectons un attribut `id` ou `name` à un bean interne, il est ignoré.

```
<bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator">
    <property name="initial" value="100000" />
    <property name="suffix" value="A" />
    <property name="prefixGenerator">
        <bean class="com.apress.springrecipes.sequence.DatePrefixGenerator">
            <property name="pattern" value="yyyyMMdd" />
        </bean>
    </property>
</bean>
```

Un bean interne peut également être déclaré dans un argument de constructeur.

```
<bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator">
    <constructor-arg>
        <bean class="com.apress.springrecipes.sequence.DatePrefixGenerator">
            <property name="pattern" value="yyyyMMdd" />
        </bean>
    </constructor-arg>
    <property name="initial" value="100000" />
    <property name="suffix" value="A" />
</bean>
```

3.5 Contrôler les propriétés par vérification des dépendances

Problème

Dans une application réelle, des centaines ou des milliers de beans peuvent être déclarés dans le conteneur IoC, avec des dépendances souvent très complexes. Avec l'injection par mutateur, il est impossible de certifier qu'une propriété sera injectée. Il est donc très difficile de contrôler que toutes les propriétés requises sont définies.

Solution

Spring offre une fonction de vérification des dépendances qui peut nous aider à contrôler si toutes les propriétés d'un certain type ont été définies pour un bean. Il suffit de préciser le mode de vérification des dépendances dans l'attribut `dependency-check` de l'élément `<bean>`. Cette fonctionnalité vérifie uniquement si les propriétés ont été définies, non si leur valeur est différente de `null`. Le Tableau 3.1 recense tous les modes de vérification des dépendances reconnus par Spring.

Tableau 3.1 : Modes de vérification des dépendances reconnus par Spring

Mode	*Description*
`none`[1]	Aucune vérification des dépendances n'est effectuée. Des propriétés peuvent rester indéfinies.
`simple`	Si une propriété de type simple (types primitifs et collections) n'a pas été fixée, une exception `UnsatisfiedDependencyException` est lancée.
`objects`	Si une propriété de type objet (autre que les types simples) n'a pas été fixée, une exception `UnsatisfiedDependencyException` est lancée.
`all`	Si une propriété de n'importe quel type n'a pas été fixée, une exception `UnsatisfiedDependencyException` est lancée.

1. Le mode par défaut est `none`, mais il est possible de le changer à l'aide de l'attribut `default-dependency-check` de l'élément racine `<beans>`. Ce mode par défaut est écrasé par tout mode précisé sur un bean. Cet attribut doit être employé avec prudence car il affecte le mode de vérification des dépendances par défaut pour tous les beans du conteneur IoC.

Explications

Contrôler les propriétés de type simple

Supposons que la propriété `suffix` n'ait pas été fixée par le générateur séquentiel. Dans ce cas, le générateur produit des numéros dont le suffixe est la chaîne de caractères `null`. Ce type de problème est souvent très difficile à déboguer, en particulier si le bean est complexe. Heureusement, Spring est capable de vérifier si toutes les propriétés d'un certain type ont été fixées. Pour demander à Spring de contrôler les propriétés de type simple (c'est-à-dire les types primitifs et les collections), nous affectons la valeur `simple` à l'attribut `dependency-check` de l'élément `<bean>`.

```
<bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator"
    dependency-check="simple">
    <property name="initial" value="100000" />
    <property name="prefixGenerator" ref="datePrefixGenerator" />
</bean>
```

Si une propriété de type simple n'a pas été fixée, une exception `UnsatisfiedDependencyException` est lancée et précise la propriété en cause.

```
Exception in thread "main"
org.springframework.beans.factory.UnsatisfiedDependencyException: Error
creating bean with name 'sequenceGenerator' defined in class path resource
[beans.xml]: Unsatisfied dependency expressed through bean property 'suffix':
Set this property value or disable dependency checking for this bean.
```

Contrôler les propriétés de type objet

Si le générateur du préfixe n'est pas défini, l'exception `NullPointerException` est lancée au moment de la demande de génération du préfixe. Pour activer la vérification des dépendances sur les propriétés de type objet (c'est-à-dire non de type simple), nous affectons la valeur `objects` à l'attribut `dependency-check`.

```
<bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator"
    dependency-check="objects">
    <property name="initial" value="100000" />
    <property name="suffix" value="A" />
</bean>
```

Dans ce cas, Spring signale, lors de l'exécution de l'application, que la propriété `prefixGenerator` n'a pas été fixée.

```
Exception in thread "main"
org.springframework.beans.factory.UnsatisfiedDependencyException: Error
creating bean with name 'sequenceGenerator' defined in class path resource
[beans.xml]: Unsatisfied dependency expressed through bean property
'prefixGenerator': Set this property value or disable dependency checking for
this bean.
```

Contrôler les propriétés de n'importe quel type

Pour contrôler toutes les propriétés de bean, quel que soit leur type, nous fixons l'attribut `dependency-check` à `all`.

```
<bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator"
    dependency-check="all">
    <property name="initial" value="100000" />
</bean>
```

Vérification des dépendances et injection par constructeur

La fonctionnalité de vérification des dépendances de Spring contrôle uniquement si une propriété a été injectée par l'intermédiaire d'un mutateur. Par conséquent, même si le générateur de préfixe a été injecté *via* un constructeur, une exception `UnsatisfiedDependencyException` est lancée.

```
<bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator"
    dependency-check="all">
    <constructor-arg ref="datePrefixGenerator" />
    <property name="initial" value="100000" />
    <property name="suffix" value="A" />
</bean>
```

3.6 Contrôler les propriétés avec l'annotation *@Required*

Problème

La fonctionnalité de vérification des dépendances de Spring ne peut contrôler que les propriétés qui sont d'un certain type. Elle n'est pas suffisamment souple pour contrôler des propriétés spécifiques. Dans la plupart des cas, nous souhaitons contrôler si certaines propriétés ont été fixées, non toutes les propriétés d'un certain type.

Solution

`RequiredAnnotationBeanPostProcessor` est un postprocesseur de beans Spring qui vérifie si toutes les propriétés de bean annotées par `@Required` ont été fixées. Un *postprocesseur de beans* est une sorte de bean Spring spécial capable d'effectuer des opérations supplémentaires sur chaque bean avant son initialisation. Pour l'activer, nous devons l'enregistrer dans le conteneur Spring IoC. Il peut uniquement contrôler si les propriétés ont été fixées, non si leur valeur est différente de `null`.

Explications

Supposons que les propriétés `prefixGenerator` et `suffix` soient obligatoires pour un générateur séquentiel. Nous pouvons alors annoter leur mutateur avec `@Required`.

```
package com.apress.springrecipes.sequence;

import org.springframework.beans.factory.annotation.Required;

public class SequenceGenerator {

    private PrefixGenerator prefixGenerator;
    private String suffix;
    ...
    @Required
    public void setPrefixGenerator(PrefixGenerator prefixGenerator) {
        this.prefixGenerator = prefixGenerator;
    }

    @Required
    public void setSuffix(String suffix) {
        this.suffix = suffix;
    }
    ...
}
```

Pour demander à Spring de contrôler si ces propriétés ont été fixées pour toutes les instances du générateur séquentiel, nous devons enregistrer une instance de `RequiredAnnotationBeanPostProcessor` dans le conteneur IoC. Si nous utilisons une fabrique de beans, l'enregistrement de ce postprocesseur doit se faire au travers de l'API. Sinon il suffit de déclarer une instance de ce postprocesseur dans le contexte d'application.

```
<bean class="org.springframework.beans.factory.annotation.➥
    RequiredAnnotationBeanPostProcessor" />
```

Avec Spring 2.5, nous pouvons simplement inclure l'élément `<context:annotation-config>` dans le fichier de configuration des beans pour qu'une instance de `RequiredAnnotationBeanPostProcessor` soit automatiquement enregistrée.

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-2.5.xsd">

    <context:annotation-config />
    ...
</beans>
```

Dans l'éventualité où des propriétés marquées par `@Required` n'ont pas été fixées, une exception `BeanInitializationException` est lancée par ce postprocesseur de beans.

```
Exception in thread "main"
org.springframework.beans.factory.BeanCreationException: Error creating bean
with name 'sequenceGenerator' defined in class path resource [beans.xml]:
Initialization of bean failed; nested exception is
org.springframework.beans.factory.BeanInitializationException: Property
'prefixGenerator' is required for bean 'sequenceGenerator'
```

Outre l'annotation `@Required`, `RequiredAnnotationBeanPostProcessor` peut également contrôler les propriétés marquées par des annotations personnalisées. Créons, par exemple, l'annotation suivante :

```
package com.apress.springrecipes.sequence;
...
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Mandatory {
}
```

Nous pouvons ensuite l'appliquer aux mutateurs des propriétés obligatoires.

```
package com.apress.springrecipes.sequence;

public class SequenceGenerator {

    private PrefixGenerator prefixGenerator;
    private String suffix;
    ...
    @Mandatory
    public void setPrefixGenerator(PrefixGenerator prefixGenerator) {
        this.prefixGenerator = prefixGenerator;
    }
```

```
    @Mandatory
    public void setSuffix(String suffix) {
        this.suffix = suffix;
    }
    ...
}
```

Pour contrôler les propriétés ayant cette annotation, nous devons l'indiquer dans la propriété `requiredAnnotationType` de `RequiredAnnotationBeanPostProcessor`.

```
<bean class="org.springframework.beans.factory.annotation.➥
    RequiredAnnotationBeanPostProcessor">
    <property name="requiredAnnotationType">
        <value>com.apress.springrecipes.sequence.Mandatory</value>
    </property>
</bean>
```

3.7 Lier automatiquement des beans par configuration XML

Problème

Lorsqu'un bean doit accéder à un autre bean, nous pouvons les lier en précisant explicitement la référence. Toutefois, si le conteneur pouvait lier automatiquement nos beans, nous ferions l'économie de la configuration manuelle des dépendances.

Solution

Le conteneur Spring IoC peut nous aider à lier automatiquement des beans. Il suffit d'indiquer le mode de liaison automatique dans l'attribut `autowire` de l'élément `<bean>`. Le Tableau 3.2 recense les modes de liaison automatique reconnus par Spring.

Tableau 3.2 : Modes de liaison automatique reconnus par Spring

Mode	*Description*
`no`[1]	Aucune liaison automatique n'est réalisée. Les dépendances doivent être établies explicitement.
`byName`	Pour chaque propriété du bean, lier un bean dont le nom correspond à celui de la propriété.
`byType`	Pour chaque propriété du bean, lier un bean dont le type est compatible avec celui de la propriété. Si plusieurs beans sont trouvés, une exception `UnsatisfiedDependencyException` est lancée.
`constructor`	Pour chaque argument de chaque constructeur, commencer par trouver un bean dont le type est compatible avec celui de l'argument. Ensuite, retenir le constructeur ayant le plus grand nombre d'arguments correspondants. En cas d'ambiguïté, une exception `UnsatisfiedDependencyException` est lancée.

Tableau 3.2 : Modes de liaison automatique reconnus par Spring *(suite)*

Mode	*Description*
`autodetect`	Si un constructeur par défaut sans argument est trouvé, les dépendances sont injectées automatiquement en fonction du type. Sinon elles sont établies automatiquement par constructeur.

1. Le mode par défaut est `no`, mais il est possible de changer cette configuration en fixant l'attribut `default-autowire` de l'élément racine `<beans>`. Le mode par défaut est écrasé par le mode défini explicitement sur un bean.

Bien que la fonctionnalité de liaison automatique soit très puissante, elle a pour inconvénient de diminuer la lisibilité de la configuration des beans. Puisque la liaison automatique est effectuée par Spring au moment de l'exécution, nous ne pouvons pas déduire les liaisons des beans à partir du fichier de configuration. En pratique, il est conseillé de mettre en place la liaison automatique uniquement dans les applications dont les dépendances entre composants restent simples.

Explications

Liaison automatique par type

Nous pouvons fixer l'attribut `autowire` du bean `sequenceGenerator` à la valeur `byType` et laisser la propriété `prefixGenerator` non fixée. Spring tente alors de lier un bean dont le type est compatible avec `PrefixGenerator`. Dans ce cas, le bean `datePrefixGenerator` sera lié automatiquement.

```
<beans ...>
    <bean id="sequenceGenerator"
        class="com.apress.springrecipes.sequence.SequenceGenerator"
        autowire="byType">
        <property name="initial" value="100000" />
        <property name="suffix" value="A" />
    </bean>

    <bean id="datePrefixGenerator"
        class="com.apress.springrecipes.sequence.DatePrefixGenerator">
        <property name="pattern" value="yyyyMMdd" />
    </bean>
</beans>
```

La liaison automatique par type présente un problème majeur : plusieurs beans du conteneur IoC peuvent être compatibles avec le type cible. Dans ce cas, Spring n'est pas en mesure de choisir le bean adapté à la propriété et ne peut donc pas effectuer de liaison automatique. Par exemple, s'il existe un autre générateur qui utilise l'année courante comme préfixe, la liaison automatique par type ne fonctionne pas.

```
<beans ...>
    <bean id="sequenceGenerator"
        class="com.apress.springrecipes.sequence.SequenceGenerator"
        autowire="byType">
        <property name="initial" value="100000" />
        <property name="suffix" value="A" />
    </bean>

    <bean id="datePrefixGenerator"
        class="com.apress.springrecipes.sequence.DatePrefixGenerator">
        <property name="pattern" value="yyyyMMdd" />
    </bean>

    <bean id="yearPrefixGenerator"
        class="com.apress.springrecipes.sequence.DatePrefixGenerator">
        <property name="pattern" value="yyyy" />
    </bean>
</beans>
```

Spring lance une exception `UnsatisfiedDependencyException` lorsque plusieurs beans peuvent servir à la liaison automatique.

```
Exception in thread "main"
org.springframework.beans.factory.UnsatisfiedDependencyException: Error
creating bean with name 'sequenceGenerator' defined in class path resource
[beans.xml]: Unsatisfied dependency expressed through bean property
'prefixGenerator': No unique bean of type
[com.apress.springrecipes.sequence.PrefixGenerator] is defined: expected single
matching bean but found 2: [datePrefixGenerator, yearPrefixGenerator]
```

Liaison automatique par nom

Le mode de liaison automatique `byName` permet parfois de résoudre les problèmes de la liaison automatique `byType`. Il fonctionne de manière comparable à la liaison par type, mais, dans ce cas, Spring tente de lier un bean de même nom à la place d'un bean de type compatible. Puisque les noms des beans sont uniques au sein d'un conteneur, la liaison automatique par nom n'est pas ambiguë.

```
<beans ...>
    <bean id="sequenceGenerator"
        class="com.apress.springrecipes.sequence.SequenceGenerator"
        autowire="byName">
        <property name="initial" value="100000" />
        <property name="suffix" value="A" />
    </bean>

    <bean id="prefixGenerator"
        class="com.apress.springrecipes.sequence.DatePrefixGenerator">
        <property name="pattern" value="yyyyMMdd" />
    </bean>
</beans>
```

Toutefois, ce mode ne fonctionne pas dans tous les cas. Il est parfois impossible que le nom du bean cible soit le même que celui de la propriété. En pratique, il est souvent nécessaire de préciser explicitement les dépendances ambiguës, tout en établissant automatiquement les autres. Autrement dit, il faut combiner les liaisons explicites et les liaisons automatiques.

Liaison automatique par constructeur

La liaison automatique par constructeur fonctionne de manière semblable au mode `byType`, mais elle est plus complexe. Dans le cas d'un bean avec un seul constructeur, Spring tente de lier à chaque argument du constructeur un bean dont le type est compatible avec celui de cet argument. En revanche, dans le cas d'un bean avec plusieurs constructeurs, le processus est plus complexe. Spring commence par rechercher un bean dont le type est compatible avec celui de chaque argument de chaque constructeur. Ensuite, il choisit le constructeur qui présente le plus grand nombre d'arguments correspondants.

Supposons que `SequenceGenerator` ait un constructeur par défaut et un constructeur avec un argument de type `PrefixGenerator`.

```
package com.apress.springrecipes.sequence;

public class SequenceGenerator {

    public SequenceGenerator() {}

    public SequenceGenerator(PrefixGenerator prefixGenerator) {
        this.prefixGenerator = prefixGenerator;
    }
    ...
}
```

Dans ce cas, le deuxième constructeur est retenu car Spring peut trouver un bean dont le type est compatible avec `PrefixGenerator`.

```
<beans ...>
    <bean id="sequenceGenerator"
        class="com.apress.springrecipes.sequence.SequenceGenerator"
        autowire="constructor">
        <property name="initial" value="100000" />
        <property name="suffix" value="A" />
    </bean>

    <bean id="datePrefixGenerator"
        class="com.apress.springrecipes.sequence.DatePrefixGenerator">
        <property name="pattern" value="yyyyMMdd" />
    </bean>
</beans>
```

Cependant, les multiples constructeurs d'une classe peuvent créer une ambiguïté dans la correspondance de leurs arguments. La situation devient encore plus complexe si nous demandons à Spring de déterminer un constructeur à notre place. Lorsque ce mode de liaison automatique est utilisé, il faut faire très attention à éviter les ambiguïtés.

Liaison automatique par détection automatique

Le mode `autodetect` demande à Spring de choisir lui-même entre les modes de liaison automatique `byType` et `constructor`. Si un constructeur par défaut sans argument est trouvé pour le bean, Spring choisit `byType`. Sinon il opte pour le mode `constructor`. Puisque la classe `SequenceGenerator` définit un constructeur par défaut, la liaison automatique par type est choisie. Autrement dit, le générateur de préfixe est injecté *via* le mutateur.

```
<beans ...>
    <bean id="sequenceGenerator"
        class="com.apress.springrecipes.sequence.SequenceGenerator"
        autowire="autodetect">
        <property name="initial" value="100000" />
        <property name="suffix" value="A" />
    </bean>

    <bean id="datePrefixGenerator"
        class="com.apress.springrecipes.sequence.DatePrefixGenerator">
        <property name="pattern" value="yyyyMMdd" />
    </bean>
</beans>
```

Liaison automatique et vérification des dépendances

Nous l'avons vu, lorsque Spring trouve plusieurs beans candidats pour la liaison automatique, il lance une exception `UnsatisfiedDependencyException`. Par ailleurs, si le mode est fixé à `byName` ou à `byType` et si Spring ne trouve pas de bean adéquat, il laisse la propriété telle quelle, ce qui peut provoquer une exception `NullPointerException` ou conduire à une valeur non initialisée. Si nous le souhaitons, nous pouvons être informés de l'échec de la liaison automatique. Pour cela, il suffit d'affecter la valeur `objects` ou `all` à l'attribut `dependency-check`. Une exception `UnsatisfiedDependencyException` est alors lancée si la liaison automatique échoue.

```
<bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator"
    autowire="byName" dependency-check="objects">
    <property name="initial" value="100000" />
    <property name="suffix" value="A" />
</bean>
```

3.8 Lier automatiquement des beans avec *@Autowired* et *@Resource*

Problème

La liaison automatique fondée sur l'attribut `autowire` dans le fichier de configuration des beans établit des dépendances pour toutes les propriétés d'un bean. Elle ne permet pas de lier uniquement certaines propriétés. Par ailleurs, il n'est possible de lier automatiquement des beans que par le type ou le nom. Lorsque aucune de ces stratégies ne répond à nos besoins, nous devons lier les beans explicitement.

Solution

Spring 2.5 apporte de nombreuses améliorations à la fonctionnalité de liaison automatique. Nous pouvons lier automatiquement une propriété en plaçant une annotation `@Autowired` ou `@Resource` sur un mutateur, un constructeur, un champ ou même une méthode. Ces annotations sont définies dans la JSR-250, *Common Annotations for the Java Platform*. Autrement dit, nous ne sommes pas réduits à l'attribut `autowire` pour satisfaire nos exigences. Toutefois, cette solution nous oblige à utiliser Java 1.5 ou une version ultérieure.

Explications

Pour demander à Spring de lier automatiquement les propriétés de bean marquées par `@Autowired` ou `@Resource`, nous devons enregistrer une instance de `AutowiredAnnotationBeanPostProcessor` dans le conteneur IoC. Si nous utilisons une fabrique de beans, l'enregistrement de ce postprocesseur doit se faire au travers de l'API. Sinon il nous suffit d'en déclarer une instance dans le contexte d'application.

```
<bean class="org.springframework.beans.factory.annotation.➥
    AutowiredAnnotationBeanPostProcessor" />
```

Ou bien nous pouvons simplement inclure l'élément `<context:annotation-config>` dans le fichier de configuration des beans pour qu'une instance de `AutowiredAnnotationBeanPostProcessor` soit automatiquement enregistrée.

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-2.5.xsd">

    <context:annotation-config />
    ...
</beans>
```

Liaison automatique d'un seul bean de type compatible

L'annotation `@Autowired` appliquée à une certaine propriété permet à Spring de la lier automatiquement. Par exemple, si nous annotons le mutateur de la propriété `prefixGenerator` avec `@Autowired`, Spring tente de lier un bean dont le type est compatible avec `PrefixGenerator`.

```
package com.apress.springrecipes.sequence;

import org.springframework.beans.factory.annotation.Autowired;

public class SequenceGenerator {
    ...
    @Autowired
    public void setPrefixGenerator(PrefixGenerator prefixGenerator) {
        this.prefixGenerator = prefixGenerator;
    }
}
```

Si un bean dont le type est compatible avec `PrefixGenerator` est défini dans le conteneur IoC, il est automatiquement affecté à la propriété `prefixGenerator`.

```
<beans ...>
    ...
    <bean id="sequenceGenerator"
        class="com.apress.springrecipes.sequence.SequenceGenerator">
        <property name="initial" value="100000" />
        <property name="suffix" value="A" />
    </bean>

    <bean id="datePrefixGenerator"
        class="com.apress.springrecipes.sequence.DatePrefixGenerator">
        <property name="pattern" value="yyyyMMdd" />
    </bean>
</beans>
```

Par défaut, toutes les propriétés marquées par `@Autowired` sont obligatoires. Lorsque Spring n'est pas en mesure de trouver un bean adapté, il lance une exception. Pour que la liaison d'une propriété soit facultative, nous devons affecter la valeur `false` à l'attribut `required` de `@Autowired`. Dans ce cas, si Spring ne trouve aucun bean adéquat, il laisse la propriété telle quelle.

```
package com.apress.springrecipes.sequence;

import org.springframework.beans.factory.annotation.Autowired;

public class SequenceGenerator {
    ...
    @Autowired(required = false)
    public void setPrefixGenerator(PrefixGenerator prefixGenerator) {
        this.prefixGenerator = prefixGenerator;
    }
}
```

L'annotation `@Autowired` s'applique non seulement à un mutateur, mais également à un constructeur. Spring tente alors de trouver un bean de type compatible pour chaque argument du constructeur.

```
package com.apress.springrecipes.sequence;

import org.springframework.beans.factory.annotation.Autowired;

public class SequenceGenerator {
    ...
    @Autowired
    public SequenceGenerator(PrefixGenerator prefixGenerator) {
        this.prefixGenerator = prefixGenerator;
    }
}
```

L'annotation `@Autowired` peut également être appliquée à un champ, même s'il n'est pas déclaré `public`. Nous pouvons ainsi omettre la déclaration d'un mutateur ou d'un constructeur pour ce champ. Spring injecte le bean correspondant dans ce champ par l'intermédiaire du mécanisme de réflexion. Toutefois, l'annotation d'un champ non public avec `@Autowired` diminue les possibilités de test du code car il devient réfractaire aux tests unitaires.

```
package com.apress.springrecipes.sequence;

import org.springframework.beans.factory.annotation.Autowired;

public class SequenceGenerator {

    @Autowired
    private PrefixGenerator prefixGenerator;
    ...
}
```

Nous pouvons même appliquer l'annotation `@Autowired` à une méthode de nom quelconque et au nombre d'arguments quelconque. Spring tente alors de lier un bean de type compatible pour chacun des arguments de la méthode.

```
package com.apress.springrecipes.sequence;

import org.springframework.beans.factory.annotation.Autowired;

public class SequenceGenerator {
    ...
    @Autowired
    public void inject(PrefixGenerator prefixGenerator) {
        this.prefixGenerator = prefixGenerator;
    }
}
```

Liaison automatique de tous les beans de type compatible

Lorsque l'annotation `@Autowired` est appliquée à une propriété de type tableau, Spring lie automatiquement tous les beans correspondants. Par exemple, si nous marquons une propriété `PrefixGenerator[]` avec `@Autowired`, Spring lie automatiquement et en une seule fois tous les beans dont le type est compatible avec `PrefixGenerator`.

```
package com.apress.springrecipes.sequence;

import org.springframework.beans.factory.annotation.Autowired;

public class SequenceGenerator {

    @Autowired
    private PrefixGenerator[] prefixGenerators;
    ...
}
```

Si plusieurs beans de type compatible avec `PrefixGenerator` sont définis dans le conteneur IoC, ils sont ajoutés automatiquement au tableau `prefixGenerators`.

```
<beans ...>
    ...
    <bean id="datePrefixGenerator"
        class="com.apress.springrecipes.sequence.DatePrefixGenerator">
        <property name="pattern" value="yyyyMMdd" />
    </bean>

    <bean id="yearPrefixGenerator"
        class="com.apress.springrecipes.sequence.DatePrefixGenerator">
        <property name="pattern" value="yyyy" />
    </bean>
</beans>
```

De la même manière, nous pouvons appliquer l'annotation `@Autowired` à une collection typée. Spring est capable de lire l'information de type de cette collection et de lier automatiquement tous les beans de type compatible.

```
package com.apress.springrecipes.sequence;

import org.springframework.beans.factory.annotation.Autowired;

public class SequenceGenerator {

    @Autowired
    private List<PrefixGenerator> prefixGenerators;
    ...
}
```

Lorsque Spring constate que l'annotation `@Autowired` est appliquée à un `java.util.Map` sécurisé dont les clés sont des chaînes de caractères, il ajoute à ce `Map` tous les beans de type compatible et utilise leur nom comme clé.

```
package com.apress.springrecipes.sequence;

import org.springframework.beans.factory.annotation.Autowired;

public class SequenceGenerator {

    @Autowired
    private Map<String, PrefixGenerator> prefixGenerators;
    ...
}
```

Liaison automatique par type avec qualificateur

Par défaut, la liaison automatique par type ne fonctionne pas lorsque plusieurs beans de type compatible existent dans le conteneur IoC. Toutefois, Spring nous permet de désigner un bean candidat en fournissant son nom dans l'annotation `@Qualifier`.

```
package com.apress.springrecipes.sequence;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;

public class SequenceGenerator {

    @Autowired
    @Qualifier("datePrefixGenerator")
    private PrefixGenerator prefixGenerator;
    ...
}
```

Spring tente alors de trouver dans le conteneur IoC un bean ayant ce nom et le lie à la propriété.

```
<bean id="datePrefixGenerator"
    class="com.apress.springrecipes.sequence.DatePrefixGenerator">
    <property name="pattern" value="yyyyMMdd" />
</bean>
```

L'annotation `@Qualifier` s'applique également à un argument de méthode.

```
package com.apress.springrecipes.sequence;ces

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;

public class SequenceGenerator {
    ...
    @Autowired
    public void inject(
          @Qualifier("datePrefixGenerator") PrefixGenerator prefixGenerator) {
        this.prefixGenerator = prefixGenerator;
    }
}
```

Nous pouvons également créer notre propre type d'annotation de qualificateur qui sera utilisée dans la liaison automatique. Ce type d'annotation doit lui-même être marqué avec `@Qualifier`.

```
package com.apress.springrecipes.sequence;
...
import org.springframework.beans.factory.annotation.Qualifier;

@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD, ElementType.PARAMETER })
@Qualifier
public @interface Generator {
    String value();
}
```

Ensuite, nous pouvons appliquer cette annotation à une propriété de bean marquée par `@Autowired`. Elle demande à Spring de lier automatiquement le bean ayant cette annotation de qualificateur et la valeur indiquée.

```
package com.apress.springrecipes.sequence;

import org.springframework.beans.factory.annotation.Autowired;

public class SequenceGenerator {

    @Autowired
    @Generator("prefix")
    private PrefixGenerator prefixGenerator;
    ...
}
```

Nous devons donner ce qualificateur au bean cible qui doit être lié automatiquement à la propriété précédente. Le qualificateur est ajouté par l'élément `<qualifier>` avec l'attribut `type`. La valeur du qualificateur est précisée dans l'attribut `value`.

```
<bean id="datePrefixGenerator"
    class="com.apress.springrecipes.sequence.DatePrefixGenerator">
    <qualifier type="Generator" value="prefix" />
    <property name="pattern" value="yyyyMMdd" />
</bean>
```

Liaison automatique par nom

Si nous souhaitons lier automatiquement les propriétés d'un bean par le nom, nous pouvons marquer un mutateur, un constructeur ou un champ avec l'annotation `@Resource` de la JSR-250. Par défaut, Spring tente de trouver un bean dont le nom correspond à celui de la propriété. Mais nous pouvons préciser explicitement le nom du bean avec l'attribut `name`.

INFO

Pour utiliser les annotations de la JSR-250, vous devez inclure `common-annotations.jar` (situé dans le répertoire `lib/j2ee` de l'installation de Spring) dans votre chemin d'accès aux classes. Toutefois, si votre application s'exécute sous Java SE 6 ou Java EE 5, il est inutile d'inclure ce fichier JAR.

```
package com.apress.springrecipes.sequence;

import javax.annotation.Resource;

public class SequenceGenerator {

    @Resource(name = "datePrefixGenerator")
    private PrefixGenerator prefixGenerator;
    ...
}
```

3.9 Hériter de la configuration d'un bean

Problème

Lors de la configuration des beans dans le conteneur Spring IoC, il est possible que plusieurs beans partagent certains éléments de configuration, comme des propriétés et des attributs de bean dans l'élément `<bean>`. Ces éléments de configuration doivent être répétés pour de multiples beans.

Solution

Spring permet d'extraire les éléments de configuration communs de manière à constituer un *bean parent*. Les beans qui héritent de ce bean parent sont appelés *beans enfants*. Les beans enfants héritent des configurations définies dans le bean parent, y compris les propriétés et les attributs de bean de l'élément `<bean>`. Cela permet d'éviter la redondance des configurations. Ils peuvent également écraser les configurations héritées si nécessaire.

Le bean parent peut jouer à la fois le rôle de template de configuration et d'instance de bean. Pour qu'il serve uniquement de template, sans possibilité d'en obtenir une instance, nous devons affecter la valeur `true` à l'attribut `abstract`. Spring sait alors qu'il ne doit pas instancier ce bean.

Tous les attributs définis dans l'élément `<bean>` parent ne sont pas hérités, par exemple les attributs `autowire` et `dependency-check`. Pour connaître les attributs hérités du parent, consultez la rubrique de documentation de Spring qui concerne l'héritage de bean.

Explications

Supposons que nous ayons besoin d'ajouter une nouvelle instance du générateur séquentiel, avec des valeurs `initial` et `suffix` identiques à celles existantes.

```
<beans ...>
    <bean id="sequenceGenerator"
        class="com.apress.springrecipes.sequence.SequenceGenerator">
        <property name="initial" value="100000" />
        <property name="suffix" value="A" />
        <property name="prefixGenerator" ref="datePrefixGenerator" />
    </bean>

    <bean id="sequenceGenerator1"
        class="com.apress.springrecipes.sequence.SequenceGenerator">
        <property name="initial" value="100000" />
        <property name="suffix" value="A" />
        <property name="prefixGenerator" ref="datePrefixGenerator" />
    </bean>

    <bean id="datePrefixGenerator"
        class="com.apress.springrecipes.sequence.DatePrefixGenerator">
        <property name="pattern" value="yyyyMMdd" />
    </bean>
</beans>
```

Pour éviter de dupliquer des propriétés, nous pouvons déclarer un bean de générateur séquentiel de base qui possède ces propriétés. Ensuite, il suffit que les deux générateurs séquentiels héritent de ce générateur de base pour que leurs propriétés correspondantes soient fixées automatiquement. Il est inutile de définir les attributs `class` des beans enfants s'ils sont identiques à celui du parent.

```
<beans ...>
    <bean id="baseSequenceGenerator"
        class="com.apress.springrecipes.sequence.SequenceGenerator">
        <property name="initial" value="100000" />
        <property name="suffix" value="A" />
        <property name="prefixGenerator" ref="datePrefixGenerator" />
    </bean>

    <bean id="sequenceGenerator" parent="baseSequenceGenerator" />

    <bean id="sequenceGenerator1" parent="baseSequenceGenerator" />
    ...
</beans>
```

Les propriétés héritées peuvent être modifiées par les beans enfants. Nous pouvons ajouter un générateur séquentiel enfant avec une valeur initiale différente.

```
<beans ...>
    <bean id="baseSequenceGenerator"
        class="com.apress.springrecipes.sequence.SequenceGenerator">
        <property name="initial" value="100000" />
        <property name="suffix" value="A" />
        <property name="prefixGenerator" ref="datePrefixGenerator" />
    </bean>
```

```
    <bean id="sequenceGenerator2" parent="baseSequenceGenerator">
        <property name="initial" value="200000" />
    </bean>
    ...
</beans>
```

Avec la configuration actuelle, nous pouvons obtenir une instance du bean du générateur séquentiel de base et l'utiliser. S'il doit servir uniquement de template, nous fixons l'attribut `abstract` à `true`. Spring ne crée alors plus d'instance de ce bean.

```
<bean id="baseSequenceGenerator" abstract="true"
    class="com.apress.springrecipes.sequence.SequenceGenerator">
    ...
</bean>
```

Nous pouvons également omettre la classe du bean parent et laisser les beans enfants la préciser. Ce cas survient lorsque le bean parent et les beans enfants ne font pas partie de la même hiérarchie de classes mais partagent des propriétés de même nom. L'attribut `abstract` du bean parent doit alors être fixé à `true` car ce bean ne peut pas être instancié. Ajoutons, par exemple, une nouvelle classe `ReverseGenerator` qui possède également une propriété `initial`.

```
package com.apress.springrecipes.sequence;

public class ReverseGenerator {

    private int initial;

    public void setInitial(int initial) {
        this.initial = initial;
    }
}
```

`SequenceGenerator` et `ReverseGenerator` ne dérivent pas de la même classe de base. Elles ne font pas partie de la même hiérarchie de classes, mais elles possèdent une propriété de même nom : `initial`. Pour extraire cette propriété commune, nous avons besoin d'un bean parent, `baseGenerator`, dans lequel aucun attribut `class` n'est défini.

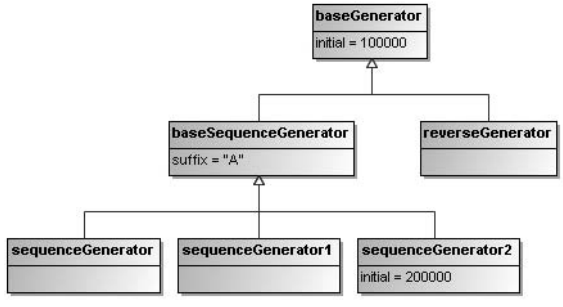
```
<beans ...>
    <bean id="baseGenerator" abstract="true">
        <property name="initial" value="100000" />
    </bean>

    <bean id="baseSequenceGenerator" abstract="true" parent="baseGenerator"
        class="com.apress.springrecipes.sequence.SequenceGenerator">
        <property name="suffix" value="A" />
        <property name="prefixGenerator" ref="datePrefixGenerator" />
    </bean>

    <bean id="reverseGenerator" parent="baseGenerator"
        class="com.apress.springrecipes.sequence.ReverseGenerator" />
```

```
    <bean id="sequenceGenerator" parent="baseSequenceGenerator" />

    <bean id="sequenceGenerator1" parent="baseSequenceGenerator" />

    <bean id="sequenceGenerator2" parent="baseSequenceGenerator">
        ...
    </bean>
    ...
</beans>
```

La Figure 3.3 présente le graphe d'objets qui correspond à cette hiérarchie de beans générateurs.

Figure 3.3

Graphe d'objets pour la hiérarchie de beans générateurs.



3.10 Affecter des collections aux propriétés de bean

Problème

Les propriétés de bean sont parfois des collections qui contiennent de multiples éléments. Nous préférerions configurer ces propriétés de type collection à partir du fichier de configuration des beans plutôt que dans le code.

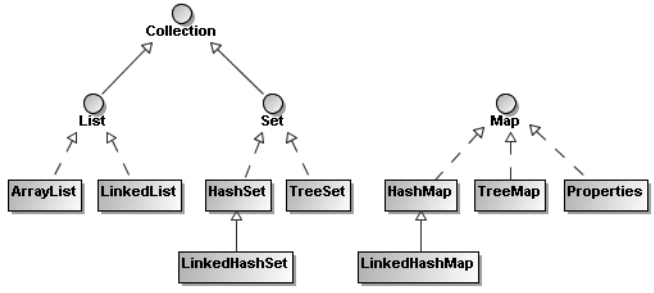
Solution

Le framework Java Collections définit un ensemble d'interfaces, d'implémentations et d'algorithmes pour différents types de collections, comme les listes (*list*), les ensembles (*set*) et les tables d'association (*map*). La Figure 3.4 montre un diagramme de classes UML simplifié qui peut faciliter la compréhension du framework Java Collections.

`List`, `Set` et `Map` sont les interfaces centrales qui représentent les trois principaux types de collections. Pour chaque type de collection, Java fournit plusieurs implémentations dont les fonctions et les caractéristiques diffèrent. Dans Spring, ces types de collections sont faciles à configurer grâce à un ensemble de balises XML intégrées, comme `<list>`, `<set>` et `<map>`.

Figure 3.4

Diagramme de classes simplifié pour le framework Java Collections.



Explications

Supposons que notre générateur séquentiel accepte plusieurs suffixes. Ils sont concaténés aux numéros de séquence en les séparant par des tirets. Nous devons accepter des suffixes ayant des types de données quelconques et les convertir en chaînes de caractères pour les ajouter aux numéros de séquence.

Listes, tableaux et ensembles

Tout d'abord, utilisons une collection `java.util.List` pour contenir nos suffixes. Une *liste* est une collection ordonnée et indexée dont les éléments sont accessibles par l'intermédiaire d'un indice ou d'une boucle for-each.

```
package com.apress.springrecipes.sequence;
...
public class SequenceGenerator {
    ...
    private List<Object> suffixes;

    public void setSuffixes(List<Object> suffixes) {
        this.suffixes = suffixes;
    }

    public synchronized String getSequence() {
        StringBuffer buffer = new StringBuffer();
        ...
        for (Object suffix : suffixes) {
            buffer.append("-");
            buffer.append(suffix);
        }
        return buffer.toString();
    }
}
```

Pour définir une propriété de type `java.util.List` dans le fichier de configuration des beans, nous employons une balise `<list>` qui contient les éléments de la liste. Ces éléments peuvent être des valeurs constantes simples (`<value>`), des références à des beans (`<ref>`), des définitions de beans internes (`<bean>`) ou des éléments `null` (`<null>`). Il est même possible d'inclure des collections dans une collection.

```
<bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator">
    <property name="prefixGenerator" ref="datePrefixGenerator" />
    <property name="initial" value="100000" />
    <property name="suffixes">
        <list>
            <value>A</value>
            <bean class="java.net.URL">
                <constructor-arg value="http" />
                <constructor-arg value="www.apress.com" />
                <constructor-arg value="/" />
            </bean>
            <null />
        </list>
    </property>
</bean>
```

Conceptuellement, un *tableau* est très semblable à une liste en cela qu'il s'agit également d'une collection ordonnée et indexée dont les éléments sont accessibles par un indice. La principale différence vient de la taille figée du tableau, qui ne peut donc pas être étendu dynamiquement. Il est possible de convertir un tableau en liste et inversement avec les méthodes `Arrays.asList()` et `List.toArray()`. Pour notre générateur séquentiel, nous pouvons utiliser un tableau `Object[]` pour contenir les suffixes et accéder à ceux-ci par un indice ou une boucle for-each.

```
package com.apress.springrecipes.sequence;
...
public class SequenceGenerator {
    ...
    private Object[] suffixes;

    public void setSuffixes(Object[] suffixes) {
        this.suffixes = suffixes;
    }
    ...
}
```

Dans le fichier de configuration des beans, la définition d'un tableau est identique à celle d'une liste et se fonde sur la balise `<list>`.

```
<bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator">
    ...
    <property name="suffixes">
        <list>
            <value>A</value>
            <bean class="java.net.URL">
                <constructor-arg value="http" />
                <constructor-arg value="www.apress.com" />
                <constructor-arg value="/" />
            </bean>
            <null />
        </list>
    </property>
</bean>
```

L'*ensemble* est un autre type de collection très répandu. La Figure 3.4 montre que `java.util.List` et `java.util.Set` étendent toutes deux l'interface `java.util.Collection`. Un ensemble diffère d'une liste en cela qu'il est ni ordonné ni indexé et qu'il ne peut contenir que des objets uniques. Autrement dit, il ne peut exister aucun élément en double dans un ensemble. Lorsque le même élément est ajouté une deuxième fois dans un ensemble, il remplace le premier. L'égalité de deux éléments est déterminée par la méthode `equals()`.

```
package com.apress.springrecipes.sequence;
...
public class SequenceGenerator {
    ...
    private Set<Object> suffixes;

    public void setSuffixes(Set<Object> suffixes) {
        this.suffixes = suffixes;
    }
    ...
}
```

Pour définir une propriété de type `java.util.Set`, nous utilisons la balise `<set>` dans laquelle les éléments sont ajoutés à la manière d'une liste.

```
<bean id="sequenceGenerator"
   class="com.apress.springrecipes.sequence.SequenceGenerator">
   ...
   <property name="suffixes">
      <set>
         <value>A</value>
         <bean class="java.net.URL">
            <constructor-arg value="http" />
            <constructor-arg value="www.apress.com" />
            <constructor-arg value="/" />
         </bean>
         <null />
      </set>
   </property>
</bean>
```

Bien que le concept d'ordre n'existe pas dans les ensembles, Spring conserve l'ordre des éléments ajoutés en utilisant un `java.util.LinkedHashSet`. Cette implémentation de l'interface `java.util.Set` préserve l'ordre des éléments.

Tables d'association et propriétés

Une *table d'association* stocke ses entrées sous forme de couples clé-valeur. L'accès à une certaine valeur de la table se fait à l'aide de la clé correspondante. Nous pouvons également parcourir les entrées de la table avec une boucle for-each. Les clés et les valeurs d'une table d'association peuvent être de type quelconque. L'égalité des clés est déterminée par la méthode `equals()`. Par exemple, nous pouvons modifier notre générateur séquentiel pour qu'il accepte une collection `java.util.Map` qui contient des suffixes avec des clés.

```
package com.apress.springrecipes.sequence;
...
public class SequenceGenerator {
    ...
    private Map<Object, Object> suffixes;

    public void setSuffixes(Map<Object, Object> suffixes) {
        this.suffixes = suffixes;
    }

    public synchronized String getSequence() {
        StringBuffer buffer = new StringBuffer();
        ...
        for (Map.Entry entry : suffixes.entrySet()) {
            buffer.append("-");
            buffer.append(entry.getKey());
            buffer.append("@");
            buffer.append(entry.getValue());
        }
        return buffer.toString();
    }
}
```

Dans Spring, une table d'association se définit à l'aide de la balise `<map>`, avec plusieurs balises `<entry>` pour le contenu. Chaque entrée est constituée d'une clé et d'une valeur. La clé est définie par la balise `<key>`. Puisque le type de la clé et celui de la valeur ne souffrent d'aucune restriction, nous pouvons employer des éléments `<value>`, `<ref>`, `<bean>` ou `<null>`. Spring préserve également l'ordre des entrées de la table en utilisant une collection `java.util.LinkedHashMap`.

```
<bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator">
    ...
    <property name="suffixes">
        <map>
            <entry>
                <key>
                    <value>type</value>
                </key>
                <value>A</value>
            </entry>
            <entry>
                <key>
                    <value>url</value>
                </key>
                <bean class="java.net.URL">
                    <constructor-arg value="http" />
                    <constructor-arg value="www.apress.com" />
                    <constructor-arg value="/" />
                </bean>
            </entry>
        </map>
    </property>
</bean>
```

Des raccourcis permettent de définir des clés et des valeurs sous forme d'attributs de la balise `<entry>`. S'il s'agit de valeurs constantes simples, nous pouvons les définir avec `key` et `value`. S'il s'agit de références de beans, nous les définissons avec `key-ref` et `value-ref`.

```
<bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator">
    ...
    <property name="suffixes">
        <map>
            <entry key="type" value="A" />
            <entry key="url">
                <bean class="java.net.URL">
                    <constructor-arg value="http" />
                    <constructor-arg value="www.apress.com" />
                    <constructor-arg value="/" />
                </bean>
            </entry>
        </map>
    </property>
</bean>
```

Une collection `java.util.Properties` ressemble fortement à une table d'association. Elle implémente également l'interface `java.util.Map` et stocke les entrées sous forme de couples clé-valeur. Toutefois, les clés et les valeurs d'une collection `Properties` sont exclusivement des chaînes de caractères.

```
package com.apress.springrecipes.sequence;
...
public class SequenceGenerator {
    ...
    private Properties suffixes;

    public void setSuffixes(Properties suffixes) {
        this.suffixes = suffixes;
    }
    ...
}
```

Pour définir une collection `java.util.Properties` dans Spring, nous employons la balise `<props>` avec plusieurs balises `<prop>` pour représenter les entrées. Chaque balise `<prop>` doit définir un attribut `key` et inclure la valeur associée.

```
<bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator">
    ...
    <property name="suffixes">
        <props>
            <prop key="type">A</prop>
            <prop key="url">http://www.apress.com/</prop>
        </props>
    </property>
</bean>
```

Fusionner avec la collection du bean parent

Lorsque les beans sont définis par héritage, la collection d'un bean enfant peut fusionner avec celle de son parent si l'attribut `merge` est fixé à `true`. Pour une collection `<list>`, les éléments enfants sont ajoutés après ceux du parent de manière à conserver l'ordre. Le générateur séquentiel suivant a donc quatre suffixes : `A`, `B`, `A` et `C`.

```
<beans ...>
    <bean id="baseSequenceGenerator"
        class="com.apress.springrecipes.sequence.SequenceGenerator">
        <property name="prefixGenerator" ref="datePrefixGenerator" />
        <property name="initial" value="100000" />
        <property name="suffixes">
            <list>
                <value>A</value>
                <value>B</value>
            </list>
        </property>
    </bean>

    <bean id="sequenceGenerator" parent="baseSequenceGenerator">
        <property name="suffixes">
            <list merge="true">
                <value>A</value>
                <value>C</value>
            </list>
        </property>
    </bean>
    ...
</beans>
```

Pour une collection `<set>` ou `<map>`, les éléments enfants remplacent ceux du parent s'ils ont la même valeur. Par conséquent, le générateur séquentiel suivant a trois suffixes : `A`, `B` et `C`.

```
<beans ...>
    <bean id="baseSequenceGenerator"
        class="com.apress.springrecipes.sequence.SequenceGenerator">
        <property name="prefixGenerator" ref="datePrefixGenerator" />
        <property name="initial" value="100000" />
        <property name="suffixes">
            <set>
                <value>A</value>
                <value>B</value>
            </set>
        </property>
    </bean>

    <bean id="sequenceGenerator" parent="baseSequenceGenerator">
        <property name="suffixes">
            <set merge="true">
                <value>A</value>
                <value>C</value>
            </set>
```

```
        </property>
    </bean>
    ...
</beans>
```

3.11 Préciser le type de données des éléments d'une collection

Problème

Par défaut, Spring considère chaque élément d'une collection comme une chaîne de caractères. Nous devons préciser le type de données des éléments de notre collection si nous ne les utilisons pas comme des chaînes.

Solution

Nous pouvons indiquer le type de données de chaque élément d'une collection à l'aide de l'attribut `type` de la balise `<value>` ou celui de l'ensemble des éléments à l'aide de l'attribut `value-type` de la balise de la collection. Avec Java 1.5 ou une version ultérieure, nous pouvons définir une collection typée afin que Spring lise les informations de type de la collection.

Explications

Supposons à présent que nous acceptions une liste de nombres entiers comme suffixes pour notre générateur séquentiel. Chaque nombre est mis en forme sur quatre chiffres par une instance de `java.text.DecimalFormat`.

```
package com.apress.springrecipes.sequence;
...
public class SequenceGenerator {
    ...
    private List<Object> suffixes;

    public void setSuffixes(List<Object> suffixes) {
        this.suffixes = suffixes;
    }

    public synchronized String getSequence() {
        StringBuffer buffer = new StringBuffer();
        ...
        DecimalFormat formatter = new DecimalFormat("0000");
        for (Object suffix : suffixes) {
            buffer.append("-");
            buffer.append(formatter.format((Integer) suffix));
        }
        return buffer.toString();
    }
}
```

Nous définissons ensuite plusieurs suffixes dans le fichier de configuration des beans.

```
<bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator">
    <property name="prefixGenerator" ref="datePrefixGenerator" />
    <property name="initial" value="100000" />
    <property name="suffixes">
        <list>
            <value>5</value>
            <value>10</value>
            <value>20</value>
        </list>
    </property>
</bean>
```

Toutefois, lors de l'exécution de cette application, nous recevons une exception `ClassCastException` qui indique que les suffixes ne peuvent pas être convertis en entiers car ils sont de type `String`. Par défaut, Spring considère chaque élément d'une collection comme une chaîne de caractères. Nous devons donc définir l'attribut `type` de la balise `<value>` pour préciser le type d'un élément.

```
<bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator">
    ...
    <property name="suffixes">
        <list>
            <value type="int">5</value>
            <value type="int">10</value>
            <value type="int">20</value>
        </list>
    </property>
</bean>
```

Ou bien, en utilisant l'attribut `value-type` de la balise de la collection, nous indiquons le type de tous ses éléments.

```
<bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator">
    ...
    <property name="suffixes">
        <list value-type="int">
            <value>5</value>
            <value>10</value>
            <value>20</value>
        </list>
    </property>
</bean>
```

Dans Java 1.5 ou une version ultérieure, nous pouvons définir notre liste `suffixes` en utilisant une collection typée qui stocke des entiers.

```
package com.apress.springrecipes.sequence;
...
public class SequenceGenerator {
    ...
    private List<Integer> suffixes;
```

```
    public void setSuffixes(List<Integer> suffixes) {
        this.suffixes = suffixes;
    }

    public synchronized String getSequence() {
        StringBuffer buffer = new StringBuffer();
        ...
        DecimalFormat formatter = new DecimalFormat("0000");
        for (int suffix : suffixes) {
            buffer.append("-");
            buffer.append(formatter.format(suffix));
        }
        return buffer.toString();
    }
}
```

En ayant défini notre collection ainsi, Spring est capable de lire les informations de type de cette collection grâce au mécanisme de réflexion. Il n'est alors plus nécessaire de préciser l'attribut `value-type` de `<list>`.

```
<bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator">
    ...
    <property name="suffixes">
        <list>
            <value>5</value>
            <value>10</value>
            <value>20</value>
        </list>
    </property>
</bean>
```

3.12 Définir des collections avec des beans de fabrique et le schéma *util*

Problème

Lorsqu'on définit des collections à l'aide des balises des collections de base, il est impossible de préciser la classe concrète d'une collection, comme `LinkedList`, `TreeSet` ou `TreeMap`. Par ailleurs, nous ne pouvons pas partager une collection entre plusieurs beans en la définissant comme un bean autonome auquel d'autres beans font référence.

Solution

Spring propose deux solutions pour dépasser les limitations des balises des collections de base. La première consiste à utiliser les beans de fabrique des collections, comme `ListFactoryBean`, `SetFactoryBean` et `MapFactoryBean`. Un *bean de fabrique* est une sorte de bean Spring particulière servant à créer un autre bean. La seconde solution consiste à utiliser les balises des collections, comme `<util:list>`, `<util:set>` et `<util:map>`, définies dans le schéma `util` apporté par Spring 2.x.

Explications

Préciser la classe concrète des collections

Nous pouvons utiliser un bean de fabrique pour définir une collection et préciser sa classe. Par exemple, spécifions la propriété `targetSetClass` de `SetFactoryBean`. Spring instancie ensuite la classe indiquée pour cette collection.

```
<bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator">
    <property name="prefixGenerator" ref="datePrefixGenerator" />
    <property name="initial" value="100000" />
    <property name="suffixes">
        <bean class="org.springframework.beans.factory.config.SetFactoryBean">
            <property name="targetSetClass">
                <value>java.util.TreeSet</value>
            </property>
            <property name="sourceSet">
                <set>
                    <value>5</value>
                    <value>10</value>
                    <value>20</value>
                </set>
            </property>
        </bean>
    </property>
</bean>
```

Nous pouvons également employer une balise de collection du schéma `util` pour définir une collection et fixer sa classe (par exemple avec l'attribut `set-class` de `<util:set>`). Il ne faut cependant pas oublier d'ajouter la définition du schéma dans l'élément racine `<beans>`.

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:util="http://www.springframework.org/schema/util"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
        http://www.springframework.org/schema/util
        http://www.springframework.org/schema/util/spring-util-2.5.xsd">

    <bean id="sequenceGenerator"
        class="com.apress.springrecipes.sequence.SequenceGenerator">
        ...
        <property name="suffixes">
            <util:set set-class="java.util.TreeSet">
                <value>5</value>
                <value>10</value>
                <value>20</value>
            </util:set>
        </property>
    </bean>
    ...
</beans>
```

Définir des collections autonomes

Les beans de fabrique des collections présentent également un autre avantage. Nous pouvons définir une collection sous forme d'un bean autonome auquel d'autres beans feront référence. Par exemple, définissons un ensemble autonome avec `SetFactoryBean`.

```
<beans ...>
    <bean id="sequenceGenerator"
        class="com.apress.springrecipes.sequence.SequenceGenerator">
        ...
        <property name="suffixes">
            <ref local="suffixes" />
        </property>
    </bean>

    <bean id="suffixes"
        class="org.springframework.beans.factory.config.SetFactoryBean">
        <property name="sourceSet">
            <set>
                <value>5</value>
                <value>10</value>
                <value>20</value>
            </set>
        </property>
    </bean>
    ...
</beans>
```

La balise `<util:set>` du schéma `util` nous permet également de définir un ensemble autonome.

```
<beans ...>
    <bean id="sequenceGenerator"
        class="com.apress.springrecipes.sequence.SequenceGenerator">
        ...
        <property name="suffixes">
            <ref local="suffixes" />
        </property>
    </bean>

    <util:set id="suffixes">
        <value>5</value>
        <value>10</value>
        <value>20</value>
    </util:set>
    ...
</beans>
```

3.13 Rechercher les composants dans le chemin d'accès aux classes

Problème

Pour que le conteneur Spring IoC prenne en charge nos composants, nous les déclarons un par un dans le fichier de configuration des beans. Nous pourrions travailler plus rapidement si Spring détectait automatiquement nos composants sans passer par une configuration manuelle.

Solution

Spring 2.5 propose une fonctionnalité puissante appelée *scan de composants* (*component scanning*). Elle peut rechercher, détecter et instancier automatiquement les composants marqués par des annotations stéréotypes et situés dans le chemin d'accès aux classes. `@Component` correspond à l'annotation de base qui signale un composant géré par Spring. `@Repository`, `@Service` et `@Controller` font partie des autres annotations stéréotypes. Elles dénotent, respectivement, des composants situés dans les couches de persistance, de service et de présentation.

Explications

Supposons qu'on nous demande de développer notre application de générateur séquentiel en utilisant des séquences provenant d'une base de données et d'enregistrer le préfixe et le suffixe de chaque séquence dans une table. Tout d'abord, créons la classe de domaine `Sequence` qui contient les propriétés `id`, `prefix` et `suffix`.

```
package com.apress.springrecipes.sequence;

public class Sequence {

    private String id;
    private String prefix;
    private String suffix;

    // Constructeurs, accesseurs et mutateurs.
    ...
}
```

Ensuite, créons une interface pour l'objet d'accès aux données (DAO, *Data Access Object*) qui prend en charge les accès à la base de données. La méthode `getSequence()` charge à partir de la table un objet `Sequence` correspondant à l'identifiant indiqué, tandis que la méthode `getNextValue()` obtient la valeur suivante pour une certaine séquence de la base de données.

```
package com.apress.springrecipes.sequence;

public interface SequenceDao {

    public Sequence getSequence(String sequenceId);
    public int getNextValue(String sequenceId);
}
```

Dans une application réelle, nous implémenterions cette interface DAO avec une technologie d'accès aux données, comme JDBC ou une correspondance objet-relationnel. Toutefois, pour nos tests, nous utilisons des tables d'association qui stockent les instances des séquences et les valeurs.

```
package com.apress.springrecipes.sequence;
...
public class SequenceDaoImpl implements SequenceDao {

    private Map<String, Sequence> sequences;
    private Map<String, Integer> values;

    public SequenceDaoImpl() {
        sequences = new HashMap<String, Sequence>();
        sequences.put("IT", new Sequence("IT", "30", "A"));
        values = new HashMap<String, Integer>();
        values.put("IT", 100000);
    }

    public Sequence getSequence(String sequenceId) {
        return sequences.get(sequenceId);
    }

    public synchronized int getNextValue(String sequenceId) {
        int value = values.get(sequenceId);
        values.put(sequenceId, value + 1);
        return value;
    }
}
```

Nous avons également besoin d'un objet, jouant le rôle de façade, pour offrir le service de génération des séquences. En interne, cet objet de service coopère avec le DAO pour traiter les demandes de génération des séquences. Il a donc besoin d'une référence au DAO.

```
package com.apress.springrecipes.sequence;

public class SequenceService {

    private SequenceDao sequenceDao;

    public void setSequenceDao(SequenceDao sequenceDao) {
        this.sequenceDao = sequenceDao;
    }
```

```
    public String generate(String sequenceId) {
        Sequence sequence = sequenceDao.getSequence(sequenceId);
        int value = sequenceDao.getNextValue(sequenceId);
        return sequence.getPrefix() + value + sequence.getSuffix();
    }
}
```

Enfin, nous devons configurer ces composants dans le fichier de configuration des beans afin que notre application fonctionne. Grâce à la liaison automatique de nos composants, nous réduisons la quantité des informations de configuration.

```
<beans ...>
    <bean id="sequenceService"
        class="com.apress.springrecipes.sequence.SequenceService"
        autowire="byType" />

    <bean id="sequenceDao"
        class="com.apress.springrecipes.sequence.SequenceDaoImpl" />
</beans>
```

Nous pouvons à présent tester ces composants à l'aide de la classe `Main` suivante :

```
package com.apress.springrecipes.sequence;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {

    public static void main(String[] args) {
        ApplicationContext context =
            new ClassPathXmlApplicationContext("beans.xml");

        SequenceService sequenceService =
                (SequenceService) context.getBean("sequenceService");

        System.out.println(sequenceService.generate("IT"));
        System.out.println(sequenceService.generate("IT"));
    }
}
```

Rechercher automatiquement des composants

Le scan de composants fourni par Spring 2.5 recherche, détecte et instancie automatiquement certains composants qui se trouvent dans le chemin d'accès aux classes. Par défaut, Spring détecte tous les composants marqués d'une annotation stéréotype. `@Component` correspond à l'annotation de base qui signale un composant géré par Spring. Nous pouvons l'appliquer à notre classe `SequenceDaoImpl`.

```
package com.apress.springrecipes.sequence;

import org.springframework.stereotype.Component;
```

```
@Component
public class SequenceDaoImpl implements SequenceDao {
    ...
}
```

Nous l'appliquons également à la classe `SequenceService` pour qu'elle soit détectée par Spring. Par ailleurs, en marquant le champ DAO avec l'annotation `@Autowired`, nous permettons à Spring de le lier automatiquement par le type.

```
package com.apress.springrecipes.sequence;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class SequenceService {

    @Autowired
    private SequenceDao sequenceDao;
    ...
}
```

Après avoir appliqué des annotations stéréotypes aux classes de nos composants, nous pouvons demander à Spring de les rechercher en déclarant un seul élément XML, `<context:component-scan>`. Dans cet élément, nous précisons le paquetage où seront recherchés nos composants. Ce paquetage et tous ses sous-paquetages sont examinés. Pour indiquer plusieurs paquetages, il suffit de les séparer par des virgules.

Cet élément enregistre également une instance de `AutowiredAnnotationBeanPostProcessor` capable de lier automatiquement les propriétés marquées par l'annotation `@Autowired`.

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-2.5.xsd">

    <context:component-scan base-package="com.apress.springrecipes.sequence" />
</beans>
```

`@Component` est l'annotation stéréotype de base pour signaler des composants d'usage général. Il existe d'autres stéréotypes spécifiques pour indiquer des composants de couches différentes. Tout d'abord, l'annotation `@Repository` désigne un composant DAO dans la couche de persistance.

```
package com.apress.springrecipes.sequence;

import org.springframework.stereotype.Repository;
```

```
@Repository
public class SequenceDaoImpl implements SequenceDao {
    ...
}
```

Quant à `@Service`, elle dénote un composant de la couche de service.

```
package com.apress.springrecipes.sequence;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class SequenceService {

    @Autowired
    private SequenceDao sequenceDao;
    ...
}
```

Le stéréotype de composant `@Controller` désigne un composant contrôleur dans la couche de présentation. Nous y reviendrons au Chapitre 10.

Filtrer les composants à rechercher

Par défaut, Spring détecte toutes les classes annotées par `@Component`, `@Repository`, `@Service`, `@Controller` ou tout type d'annotation personnalisé lui-même marqué par `@Component`. Nous pouvons personnaliser la recherche en appliquant un ou plusieurs filtres d'inclusion ou d'exclusion.

Spring reconnaît quatre types d'expressions de filtrage. Les types `annotation` et `assignable` permettent de définir un filtre fondé sur un type d'annotation et une classe ou une interface. Les types `regex` et `aspectj` permettent de préciser une expression régulière et une expression de point d'action (*pointcut*) AspectJ pour la correspondance des classes.

Par exemple, le scan de composants suivant inclut toutes les classes dont le nom contient les mots `Dao` ou `Service` et exclut celles annotées avec `@Controller` :

```
<beans ...>
    <context:component-scan base-package="com.apress.springrecipes.sequence">
        <context:include-filter type="regex"
            expression="com\.apress\.springrecipes\.sequence\..*Dao.*" />
        <context:include-filter type="regex"
            expression="com\.apress\.springrecipes\.sequence\..*Service.*" />
        <context:exclude-filter type="annotation"
            expression="org.springframework.stereotype.Controller" />
    </context:component-scan>
</beans>
```

Puisque ces filtres détectent toutes les classes dont le nom contient les mots `Dao` ou `Service`, les composants `SequenceDaoImpl` et `SequenceService` sont détectés automatiquement, même sans annotation stéréotype.

Nommer les composants détectés

Par défaut, Spring nomme les composants détectés en mettant en minuscule le premier caractère du seul nom de sa classe. Par exemple, un composant de classe `SequenceService` est nommé `sequenceService`. Nous pouvons définir explicitement le nom d'un composant en le précisant dans la valeur de l'annotation stéréotype.

```
package com.apress.springrecipes.sequence;
...
import org.springframework.stereotype.Service;

@Service("sequenceService")
public class SequenceService {
    ...
}

---

package com.apress.springrecipes.sequence;

import org.springframework.stereotype.Repository;

@Repository("sequenceDao")
public class SequenceDaoImpl implements SequenceDao {
    ...
}
```

Nous pouvons développer notre propre stratégie de nommage en implémentant l'interface `BeanNameGenerator` et en l'indiquant dans l'attribut `name-generator` de l'élément `<context:component-scan>`.

3.14 En résumé

Dans ce chapitre, nous avons étudié les bases de la configuration des beans dans le conteneur Spring IoC. Spring prend en charge plusieurs types de configurations. La configuration XML est la plus simple et la plus mûre. Spring propose deux implémentations du conteneur IoC. L'implémentation de base est la fabrique de beans, la plus élaborée est le contexte d'application. Il est préférable d'utiliser le contexte d'application, à moins que les ressources soient limitées. Pour définir des propriétés de bean, qui peuvent être de simples valeurs, des collections ou des références de beans, Spring propose l'injection par mutateur et l'injection par constructeur.

La vérification des dépendances et la liaison automatique sont deux fonctionnalités du conteneur fourni par Spring. Grâce à la vérification des dépendances, nous pouvons contrôler si toutes les propriétés requises sont fixées. Grâce à la liaison automatique, nos beans peuvent être liés automatiquement, en fonction du type, du nom ou de l'annotation. L'ancien style de configuration de ces deux fonctionnalités passe par des attributs XML. La nouvelle forme se fonde sur des annotations et des postprocesseurs de beans qui apportent une plus grande souplesse.

Spring prend en charge l'héritage de bean en déplaçant les configurations communes dans un bean parent. Celui-ci peut servir de template de configuration, d'instance de bean ou les deux à la fois.

Puisque les collections sont des éléments de programmation essentiels en Java, Spring fournit différentes balises pour que nous puissions configurer facilement des collections dans le fichier de configuration des beans. Nous pouvons employer les beans de fabrique de collections ou les balises des collections du schéma `util` pour préciser les détails d'une collection et pour définir des collections sous forme de beans autonomes partagés par plusieurs beans.

Enfin, Spring peut détecter automatiquement nos composants situés dans le chemin d'accès aux classes. Par défaut, il détecte tous les composants ayant des annotations stéréotypes particulières. Les filtres permettent d'inclure ou d'exclure certains composants. Le scan de composants est une fonctionnalité puissante qui réduit la quantité d'informations de configuration.

Le chapitre suivant présente les fonctionnalités élaborées du conteneur Spring IoC que nous n'avons pas étudiées dans ce chapitre.