Chapitre **3**

L'héritage

3.1 LE PRINCIPE DE L'HÉRITAGE

3.1.1 Les définitions

Le deuxième grand principe de la programmation objet après l'encapsulation (voir figure 2.3, page 31) est le concept d'héritage. Une classe B qui hérite d'une classe A hérite des attributs et des méthodes de la classe A sans avoir à les redéfinir. B est une **sous-classe** de A ; ou encore A est la **super-classe** de B ou la classe de base. On dit encore que B est une **classe dérivée** de la classe A, ou que la classe B étend la classe A. Une classe ne peut avoir qu'une seule super-classe ; il n'y a pas d'héritage multiple en Java. Par contre, elle peut avoir plusieurs sous-classes (voir figure 3.2).

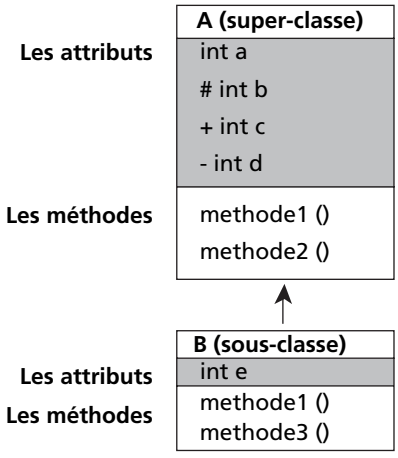

Figure 3.1 — Le principe de l'héritage : la classe B hérite de la classe A.
indique un attribut protected.

3.1.2 La redéfinition des méthodes et l'utilisation du mot-clé protected

Les droits d'accès aux données (private, public et de paquetage) ont été présentés à la page 67. Le concept d'héritage ajoute un nouveau droit d'accès dit protégé (**protected)** : celui pour une classe dérivée d'accéder aux attributs ou aux méthodes de sa super-classe qui ont été déclarés `protected`. Un attribut déclaré `protected` est accessible dans son **paquetage** et dans ses **classes dérivées**. Si un attribut de la super-classe est privé (private), la sous-classe ne peut y accéder. Sur la figure 3.1, un objet de la classe B peut accéder aux attributs a, b, c et e (a, b et c sont obtenus par héritage ; d privé est inaccessible de la classe B). La classe B contient les méthodes methode1() et methode3() de sa classe B, mais peut aussi accéder aux méthodes methode1() et methode2() de la classe A.

Soit b1 un objet de la classe B obtenu par B b1 = new B() :

accès aux attributs :

b1 peut accéder à l'attribut `c` (public : b1.c), à l'attribut `b` (protected dont accessible dans la sous-classe : b1.b), à l'attribut `a` si classe B et classe A sont dans le même paquetage (b1.a). b1 ne peut pas accéder à l'attribut `d` qui est privé (b1.d interdit). Voir page 67.

Accès aux méthodes :

- b1 peut activer une méthode spécifique de B, methode3() par exemple, comme suit : b1.methode3().
- b1 peut activer une méthode non redéfinie de sa super-classe A comme si elle faisait partie de sa classe B. Exemple : b1.methode2().
- methode1() est redéfinie dans la sous-classe B. On dit qu'il y a **redéfinition de méthodes**. Contrairement aux méthodes surchargées d'une classe qui doivent se différencier par les paramètres (voir page 46), une méthode redéfinie doit avoir un **prototype identique** à la fonction de sa super-classe. b1 exécutera methode1() de sa classe B sur l'instruction qui suit : b1.methode1(). L'objet peut aussi activer la méthode de sa super-classe A en préfixant l'appel de la méthode de **super** : **b1.super.methode1().** On suppose sur cet exemple que methode1() de la classe A et methode1() de la classe B ont le même nombre de paramètres, chacun des paramètres ayant le même type, et la valeur de retour est la même.

Un des avantages de la programmation objet est de pouvoir réutiliser et enrichir des classes sans avoir à tout redéfinir. La super-classe contient les attributs et méthodes **communs** aux classes dérivées. La classe dérivée ajoute des attributs et des méthodes qui spécialisent la super-classe. Si une méthode de la super-classe ne convient pas, on peut la redéfinir dans la sous-classe. Cette méthode redéfinie peut appeler la méthode de la super-classe si elle convient pour effectuer une partie du traitement. On peut utiliser sans les définir les méthodes de la super-classe si elles conviennent.

Remarque : si une classe est déclarée **abstraite,** on ne peut instancier de variables de cette classe. C'est un modèle pour de futures classes dérivées. On peut aussi interdire en Java la dérivation d'une classe en la déclarant `final`.

```
final class A {
  ...
}
```

3.2 LES CLASSES PERSONNE, SECRETAIRE, ENSEIGNANT, ETUDIANT

Dans un établissement d'enseignement et de manière schématique, on trouve trois sortes de personnes : des administratifs (au sens large, incluant des techniciens) représentés par la catégorie secrétaire, des enseignants et des étudiants. Chaque personne est caractérisée par son nom et prénom, son adresse (rue et ville) qui sont des attributs privés et communs à toutes les personnes. On peut représenter une personne suivant un schéma UML comme indiqué sur la figure 3.2. Les variables d'instances (attributs) sont le nom, le prénom, la rue et la ville. nbPersonnes est une variable de classe (donc static) qui comptabilise le nombre de Personne dans l'établissement. Cette variable de classe existe en un exemplaire indépendant de tout objet. Sur la figure 2.9, page 43 et sur la figure 2.11, page 48, les variables `static` apparaissent à part ; elles apparaissent en souligné avec les variables d'instance dans les classes de la figure 3.2 pour simplifier le schéma ; il est important de bien comprendre qu'elles ont un comportement différent des variables d'instance.

On définit les méthodes `public` suivantes de la classe **Personne** :

- le constructeur **Personne** (String nom, String prenom, String rue, String ville) : crée et initialise un objet de type Personne.
- String **toString** () : fournit une chaîne de caractères correspondant aux caractéristiques (attributs) d'une personne.
- **ecrirePersonne** () : pourrait écrire les caractéristiques d'une personne. Sur l'exemple ci-dessous, elle ne fait rien. Elle est déclarée **abstraite**.
- static **nbPersonnes** () : écrit le nombre total de personnes et le nombre de personnes par catégorie. C'est une méthode de classe.
- **modifier ersonne** (String rue, String ville) : modifie l'adresse d'une personne et appelle **ecrirePersonne** () pour vérifier que la modification a bien été faite.

3.2.1 La classe abstraite Personne

Si dans l'établissement, il n'y a exclusivement que trois catégories de personnes, une Personne n'existe pas. Seuls existent des secrétaires, des enseignants et des étudiants. Néanmoins, la classe Personne est utile dans la mesure où elle permet de regrouper les attributs et les méthodes communs aux trois catégories de personnel. La classe Personne est déclarée **abstraite**. On ne peut pas créer (instancier) d'objet de type Personne :

```
Personne p = new Personne ( "Dupond", "Jacques", "rue des mimosas",
                                                        "Paris");
```

fournit une **erreur** de compilation.

3.2.2 Les classes Secretaire, Enseignant et Etudiant

Une **Secretaire** est une Personne. Elle possède toutes les caractéristiques d'une Personne (nom, prenom, rue, ville) plus les caractéristiques spécifiques d'une secrétaire soit sur l'exemple, un numéro de bureau. Les méthodes suivantes sont définies pour un objet de la classe **Secretaire** :

- **Secretaire** (String nom, String prenom, String rue, String ville, String numeroBureau) : le constructeur d'un objet de la classe Secretaire doit fournir les caractéristiques pour construire une Personne, plus les spécificités de la classe Secretaire (numéro de bureau).

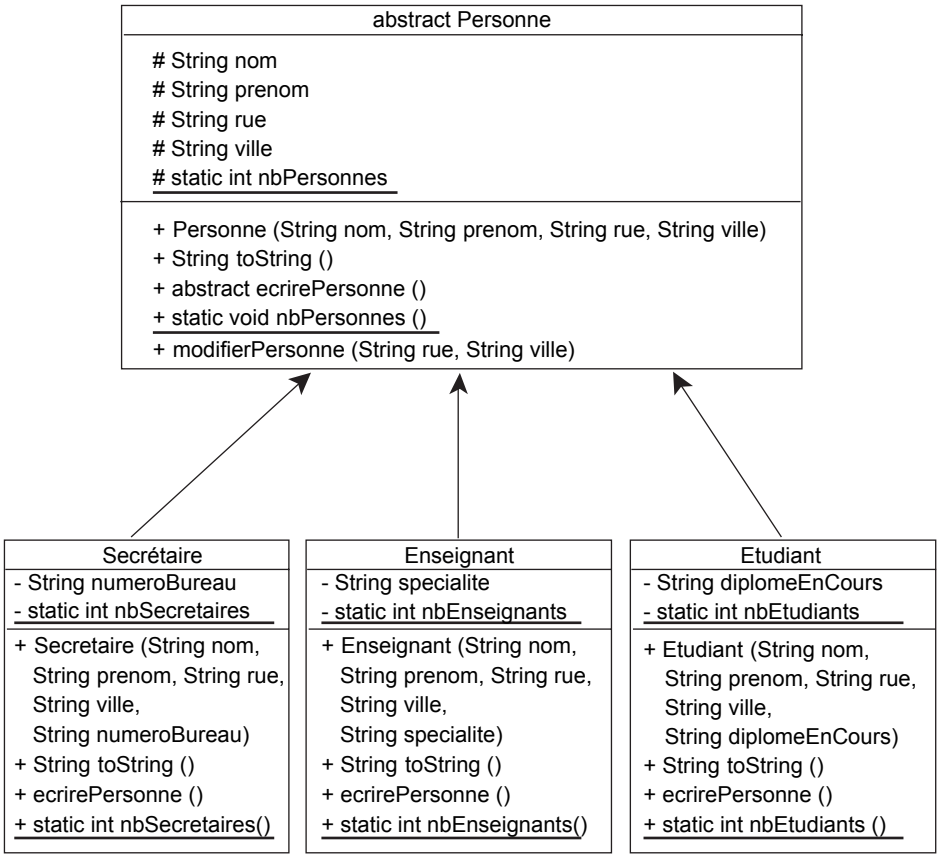

Figure 3.2 — Arbre d'héritage.

Les classes Secretaire, Enseignant et Etudiant sont des classes dérivées de la classe Personne. La classe Personne est abstraite et ne peut être instanciée.

- String **toString** () : fournit une chaîne contenant les caractéristiques d'une Secretaire.
- **ecrirePersonne** () : écrit "Secrétaire : " suivi des caractéristiques d'une Secretaire.

De même, un **Enseignant** est une Personne enseignant une spécialité (mathématiques, informatique, anglais, gestion, etc.). Un **Etudiant** est une Personne préparant un diplôme (diplomeEnCours). Les méthodes pour Enseignant et Etudiant sont similaires à celles de Secretaire. Une variable privée `static` dans chaque classe compte le nombre de personnes créées dans chaque catégorie. Une méthode `static` du même nom que la variable fournit la valeur de cette variable `static` (nbSecretaires, nbEnseignants, nbEtudiants).

3.2.3 Les mots-clés abstract, extends, super

Le programme Java correspondant à l'arbre d'héritage de la figure 3.2 est donné ci-après. Une classe contenant une méthode abstraite est forcément abstraite même si on ne l'indique pas explicitement. Ci-dessous, les attributs *protected* nom, prenom, rue, ville et la variable `static` nbPersonnes sont accessibles dans les classes dérivées Secretaire, Enseignant ou Etudiant même si les classes ne sont pas dans le même paquetage. Voir protected page 94.

```
// PPPersonne.java  Programme Principal Personne
//                  exemples d'héritage

// une personne
abstract class Personne {         // classe abstraite, non instanciable
  protected String nom;
  protected String prenom;
  protected String rue;
  protected String ville;
  protected static int nbPersonnes = 0;           // nombre de Personne

  // constructeur de Personne
  Personne (String nom, String prenom, String rue, String ville) {
    this.nom    = nom;                             // voir this page 38
    this.prenom = prenom;
    this.rue    = rue;
    this.ville  = ville;
    nbPersonnes++;
  }

  // fournir les caractéristiques d'une Personne sous forme
  // d'un objet de la classe String
  public String toString() {
    return nom + " " + prenom + " " + rue + " " + ville;
  }

  // méthode abstraite          (à redéfinir dans les classes dérivées)
  abstract void ecrirePersonne ();

  static void nbPersonnes () {
    System.out.println (
      "\nNombre d'employés     : " + nbPersonnes +
```

```
        "\nNombre de secrétaires : " + Secretaire.nbSecretaires() +
        "\nNombre d'enseignants  : " + Enseignant.nbEnseignants() +
        "\nNombre d'étudiants    : " + Etudiant.nbEtudiants()
      );
    }

    void modifierPersonne (String rue, String ville) {
      this.rue     = rue;
      this.ville   = ville;
      ecrirePersonne();               // de la classe dérivée de l'objet
    }

  }  // Personne
```

Une **Secretaire est une Personne** ayant une caractéristique supplémentaire : un numéro de bureau. La variable static nbSecretaires est incrémentée dans le constructeur de l'objet Secretaire. L'appel **super** (nom, prenom, rue, ville) fait appel au constructeur de la super-classe chargé d'initialiser les données communes à toute Personne : nom, prenom, rue et ville. De même, l'appel **super.toString()** fournit une chaîne de caractères en utilisant la méthode toString() de la super-classe de Secretaire (soit la classe Personne). Cet appel fournit une chaîne de caractères donnant les caractéristiques communes d'une personne. A cette chaîne, la méthode toString() de Secretaire ajoute le numéro de bureau qui lui est spécifique d'un objet Secretaire. Le principe est le même pour les méthodes toString() de Enseignant et Etudiant : toString() de la super-classe fournit les caractéristiques communes et toString() de la classe dérivée, les spécificités de la classe. On voit bien sur cet exemple que le "savoir-faire" de la super-classe est réutilisé sans devoir le réécrire. On en hérite et on l'enrichit.

```
class Secretaire extends Personne {               // héritage de classe
  private String numeroBureau;
  private static int nbSecretaires = 0;          // nombre de Secretaire

  Secretaire (String nom, String prenom, String rue, String ville,
              String numeroBureau) {
    // le constructeur de la super-classe
    super (nom, prenom, rue, ville);  // appel du constructeur Personne
    this.numeroBureau = numeroBureau;
    nbSecretaires++;
  }

  public String toString () {
    // super.toString() : toString() de la super-classe
    return super.toString() + "\n  N° bureau : " + numeroBureau;
  }

  void ecrirePersonne () {
    System.out.println ("Secrétaire : " + toString());
  }

  static int nbSecretaires () { return nbSecretaires; }

}  // Secretaire
```

Un **Enseignant est une Personne** enseignant une spécialité. La variable `static` nbEnseignants compte le nombre d'enseignants ; elle est incrémentée dans le constructeur Enseignant.

```
class Enseignant extends Personne {                // héritage de classe
  private String specialite;
  private static int nbEnseignants = 0;           // nombre d'Enseignant

  Enseignant (String nom, String prenom, String rue, String ville,
              String specialite) {
    super (nom, prenom, rue, ville);  // appel du constructeur Personne
    this.specialite = specialite;
    nbEnseignants++;
  }

  public String toString () {
    return super.toString() + "\n  spécialité : " + specialite;
  }

  void ecrirePersonne () {
    System.out.println ("Enseignant : " + toString());
  }

  static int nbEnseignants () { return nbEnseignants; }

} // Enseignant
```

Un **Etudiant est une P ersonne** préparant un diplôme. La variable `static` nbEtudiants compte le nombre d'étudiants ; elle est incrémentée dans le constructeur Etudiant.

```
class Etudiant extends Personne {                  // héritage de classe
  private String diplomeEnCours;
  private static int nbEtudiants = 0;               // nombre d'Etudiant

  Etudiant (String nom, String prenom, String rue, String ville,
            String diplomeEnCours) {
    super (nom, prenom, rue, ville);  // appel du constructeur Personne
    this.diplomeEnCours = diplomeEnCours;
    nbEtudiants++;
  }

  public String toString () {
    return super.toString() + "\n  Diplome en cours : "
                                                    + diplomeEnCours;
  }

  void ecrirePersonne () {
    System.out.println ("Etudiant : " + toString());
  }

  public String diplomeEnCours () {
    return diplomeEnCours;
  }

  static int nbEtudiants () { return nbEtudiants; }
```

```
} // Etudiant
```

La classe **PPPersonne** crée des objets Secretaire, Enseignant et Etudiant. La méthode static nbPersonnes() est appelée en la préfixant de la classe Personne.

```
class PPPersonne {                          // Programme Principal Personne
  public static void main(String[] args) {
    Secretaire chantal = new Secretaire ("Dupond", "Chantal",
      "rue des mimosas", "Rennes", "A123");
    chantal.ecrirePersonne();

    Enseignant michel = new Enseignant ("Martin", "Michel",
      "bd St-Antoine", "Rennes", "Maths");
    michel.ecrirePersonne();

    Etudiant e1 = new Etudiant ("Martin", "Guillaume",
      "bd St-Jacques", "Bordeaux", "licence info");
    e1.ecrirePersonne();

    Etudiant e2 = new Etudiant ("Dufour", "Stéphanie",
      "rue des saules", "Lyon", "DUT info");
    e2.ecrirePersonne();

    Personne.nbPersonnes();                          // méthode static

    System.out.println ("\n\nAprès modification :");
    chantal.modifierPersonne ("rue des sorbiers", "Nantes");
    michel.modifierPersonne ("rue des lilas", "Rennes");

  }  // main

}  // class PPPersonne
```

Exemples de résultats d'exécution de PPPersonne :

```
Secrétaire : Dupond Chantal rue des mimosas Rennes
  N° bureau : A123
Enseignant : Martin Michel bd St-Antoine Rennes
  spécialité : Maths
Etudiant : Martin Guillaume bd St-Jacques Bordeaux
  Diplome en cours : licence info
Etudiant : Dufour Stéphanie rue des saules Lyon
  Diplome en cours : DUT info

Nombre d'employés     : 4
Nombre de secrétaires : 1
Nombre d'enseignants  : 1
Nombre d'étudiants    : 2

Après modification :
Secrétaire : Dupond Chantal rue des sorbiers Nantes
  N° bureau : A123
Enseignant : Martin Michel rue des lilas Rennes
  spécialité : Maths
```

Remarque : on pourrait ne pas déclarer la méthode ecrirePersonne() abstraite en fournissant une méthode vide. Mais alors, les classes dérivées ne seraient plus obligées de la redéfinir.

3.2.4 Les méthodes abstraites, la liaison dynamique, le polymorphisme

3.2.4.1 Le polymorphisme d'une méthode redéfinie : ***ecrirePersonne()***

La méthode modifierPersonne modifie l'adresse (rue, ville) d'une Personne quelle que soit sa catégorie. Elle se trouve dans la classe commune Personne. Elle peut être appelée pour un objet Secretaire, Enseignant ou Etudiant.

```
void modifierPersonne (String rue, String ville) {
    this.rue   = rue;
    this.ville = ville;
    ecrirePersonne();       // ecrirePersonne() de la classe de l'objet
  }
```

Par contre, que fait la méthode ecrirePersonne() ? Si le compilateur faisait une **liaison statique**, il générerait des instructions prenant en compte la méthode ecrirePersonne() de la classe Personne et qui en fait ne fait rien sur cet exemple.

Quand le compilateur prévoit une **liaison dynamique** , la méthode ecrirePersonne() qui est exécutée dépend à l'exécution de la classe de l'objet. La méthode est d'abord recherchée dans la classe de l'objet, et en cas d'échec dans sa super-classe. Si chaque sous-classe redéfinit la méthode, celle de la super-classe n'est jamais exécutée et peut être déclarée abstraite. Elle sert de prototype lors de la compilation de modifierPersonne() de la classe Personne. Si la méthode est déclarée abstraite, les sous-classes doivent la redéfinir sinon, il y a un message d'erreur à la compilation.

Les instructions suivantes appellent la méthode modifierPersonne() de Personne pour deux objets différents dont la classe est dérivée de Personne.

```
chantal.modifierPersonne ("rue des sorbiers", "Nantes");
michel.modifierPersonne  ("rue des lilas", "Rennes");
```

On obtient les résultats indiqués ci-après. Pour l'objet chantal, la méthode modifierPersonne() appelle la méthode ecrirePersonne() de Secretaire. Pour l'objet michel, la méthode modifierPersonne() appelle la méthode ecrirePersonne() de Enseignant. On parle alors de **polymorphisme**. Le même appel de la méthode ecrirePersonne() de modifierPersonne() déclenche des méthodes redéfinies différentes (portant le même nom) de différentes classes dérivées. Le polymorphisme découle de la notion de redéfinition de méthodes entre une super-classe et une sous-classe.

```
Secrétaire : Dupond Chantal rue des sorbiers Nantes
  N° bureau : A123
Enseignant : Martin Michel rue des lilas Rennes
  spécialité : Maths
```

3.2.4.2 Le polymorphisme d'une variable de la super-classe

Le programme PPPersonne vu ci-dessus pourrait être écrit comme suit. La variable locale p est une référence sur une Personne. On ne peut pas créer d'instance de Personne (la classe est abstraite), mais on peut référencer à l'aide d'une variable de type Personne, un **descendant** de Personne (Secretaire, Enseignant ou Etudiant). L'instruction **p.ecrirePersonne() est polymorphe** ; la méthode appelée est celle de la classe de l'objet.

```
class PPPersonne2 {                    // Programme Principal Personne 2
  public static void main (String[] args) {
    Personne p;

    p = new Secretaire ("Dupond", "Chantal",
      "rue des mimosas", "Rennes", "A123");
    p.ecrirePersonne();                // ecrirePersonne() de Secretaire

    p = new Enseignant ("Martin", "Michel",
      "bd St-Antoine", "Rennes", "Maths");
    p.ecrirePersonne();                // ecrirePersonne() de Enseignant

    p = new Etudiant ("Martin", "Guillaume",
      "bd St-Jacques", "Bordeaux", "licence info");
    p.ecrirePersonne();                  // ecrirePersonne() de Etudiant

    p = new Etudiant ("Dufour", "Stéphanie",
      "rue des saules", "Lyon", "DUT info");
    p.ecrirePersonne();                  // ecrirePersonne() de Etudiant

    Personne.nbPersonnes();                              // méthode static
```

Peut-on à partir d'une référence de Personne, accéder aux attributs ou aux méthodes non redéfinies d'une classe dérivée ? Réponse : oui en forçant le type si l'objet est bien du type dans lequel on veut le convertir (sinon, il y a lancement d'une exception de type ClassCastException). Voir page 77.

```
    // À partir d'une référence de Personne,
    // accéder à un objet dérivé

    // p référence le dernier Etudiant créé ci-dessus.
    // Erreur de compilation : diplomeEnCours n'est pas un méthode
    // de la classe Personne
    //System.out.println ("Diplôme en cours : " + p.diplomeEnCours());

    // Erreur de compilation : impossible de convertir Personne
    // en Secretaire implicitement
    //Secretaire sec = p;

    // Erreur d'exécution : Exception ClassCastException :
    // p n'est pas une Secretaire (c'est un Etudiant)
    //Secretaire sec = (Secretaire) p;

    Etudiant et1 = (Etudiant) p;                  // OK p est un Etudiant
    System.out.println ("Diplôme en cours : " + et1.diplomeEnCours());
```

```
        // teste si p est un objet de la classe Etudiant
        if (p instanceof Etudiant) {
          System.out.println ("p est de la classe Etudiant");
        }
    }  // main
}  // PPPersonne2
```

3.2.4.3 Le polymorphisme de *toString()*

Si on remplace la méthode abstraite de la classe Personne par :

```
void ecrirePersonne () {
  System.out.println (toString());     // ou System.out.println (this);
};
```

et si on supprime ecrirePersonne() dans les sous-classes Secretaire, Enseignant et Etudiant, on obtient le résultat d'exécution ci-dessous. `System.out.println (this);` est une écriture simplifiée équivalente à `System.out.println (toString())`.

La méthode `ecrirePersonne()` appelée est celle unique de la classe `Personne` ; par contre la méthode `toString()` appelée dans ecrirePersonne() est celle de la classe de l'objet (`Secretaire`, `Enseignant` ou `Etudiant`) en raison de la liaison dynamique. Si plus tard, on définit une classe `Technicien`, la méthode `ecrirePersonne()` de `Personne` fera appel à la méthode `toString()` de `Technicien`, et ce sans aucune modification. Il suffira de définir une méthode toString() pour la classe Technicien. En l'absence de cette définition, c'est la méthode toString() de Personne qui sera utilisée. La méthode toString() de Personne est **polymorphe** puisque le même appel de méthode met en œuvre à l'exécution des méthodes dérivées redéfinies différentes suivant l'objet qui déclenche l'appel.

Résultats de l'exécution de PPPersonne2 :

ecrirePersonne() utilise :

```
Dupond Chantal rue des mimosas Rennes       toString() de Secretaire
  N° bureau : A123
Martin Michel bd St-Antoine Rennes          toString() de Enseignant
  spécialité : Maths
Martin Guillaume bd St-Jacques Bordeaux       toString() de Etudiant
  Diplome en cours : licence info
Dufour Stéphanie rue des saules Lyon
  Diplome en cours : DUT info

Nombre d'employés     : 4
Nombre de secrétaires : 1
Nombre d'enseignants  : 1
Nombre d'étudiants    : 2

Diplôme en cours : DUT info

P est de la classe Etudiant
```

Remarque : en Java, par défaut, la liaison est dynamique. Cependant, si la classe est déclarée `final`, c'est-à-dire qu'elle ne peut pas avoir de sous-classes, la liaison sera statique pour les méthodes de cette classe car la liaison statique est plus rapide. De même, une méthode peut être déclarée `final` auquel cas, elle ne pourra pas être redéfinie dans une sous-classe, et aura une liaison statique. En C++, les liaisons sont par défaut statiques. Il faut demander une liaison dynamique pour une méthode en la faisant précéder du mot-clé virtual.

Exercice 3.1 – Ajouter un nom pour une Pile

On dispose de la classe `Pile` définie en 2.10.2, page 78. Écrire une classe `PileNom` dérivée de `Pile` et comportant en plus, un nom de type `String`, un constructeur de `PileNom` et une méthode `listerPile()` qui écrit le nom de la `Pile` et liste les éléments de la pile. Écrire une classe PPPile de test.

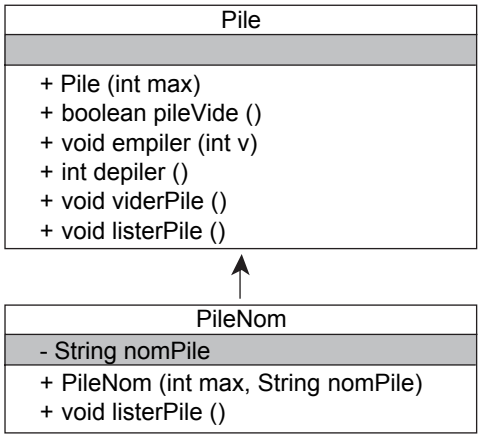

Figure 3.3 — La classe PileNom dérive de la classe Pile (héritage de classe).

3.3 LA SUPER-CLASSE OBJECT

En Java, toutes les classes dérivent implicitement d'une classe Object qui est la racine de l'arbre d'héritage. Cette classe définit une méthode toString() qui écrit le nom de la classe et un numéro d'objet. Les classes dérivées peuvent redéfinir cette méthode. Voir toString(), page 82. Puisque toute classe dérive de la classe Object, une référence de type Object peut référencer n'importe quel type d'objet : une référence de la super-classe peut être utilisée dans les classes dérivées. Voir polymorphisme d'une variable, page 103.

```
Object objet;
objet = new Secretaire ("Dupond", "Chantal",
                                  "rue des mimosas", "Rennes", "A123");
objet = new String ("Bonjour");
objet = new Complex (2, 1.50);
```

Le chapitre 4 utilise ce concept pour créer des listes d'objets (voir figure 4.3).

3.4 LA HIÉRARCHIE DES EXCEPTIONS

Les exceptions constituent un arbre hiérarchique utilisant la notion d'héritage.

```
Object                                  la classe racine de toutes les classes
  Throwable                  getMessage(), toString(), printStackTrace(), etc.
    Error                          erreurs graves, abandon du programme
    Exception                                  anomalies récupérables
        RunTimeException                          erreur à l'exécution
            ArithmeticException          division par zéro par exemple
            ClassCastException       mauvaise conversion de classes (cast)
            IndexOutOfBoundsException   en dehors des limites (tableau)
            NullPointerException                      référence nulle
            IllegalArgumentException
                NumberFormatException                 nombre mal formé
        IOException                          erreurs d'entrées-sorties
            FileNotFoundException                     fichier inconnu
```

Dans un catch, on peut capter l'exception à son niveau le plus général (Exception), ou à un niveau plus bas (FileNotFoundException par exemple) de façon à avoir un message plus approprié. On peut avoir plusieurs catch pour un même try. Voir exceptions page 74.

3.5 LES INTERFACES

Java n'autorise que l'héritage simple. Une classe ne peut avoir qu'une seule superclasse. Une certaine forme d'héritage multiple est obtenue grâce à la notion d'interface. Sur la figure 3.4, on définit de manière formelle, une classe AA qui hérite de la classe A et une classe BB qui hérite de la classe B. L'interface Modifiable permet de voir les classes AA et BB comme ayant une propriété commune, celle d'être modifiable. Une interface définit uniquement des constantes et des prototypes de méthodes.

```
interface Modifiable {
  static final int MIN = -5;
  static final int MAX = +5 ;                          // constante

  void zoomer (int n) ;                                // prototype
}  // Modifiable
```

Chacune des classes désirant être reconnues comme ayant la propriété Modifiable doit le faire savoir en implémentant l'interface Modifiable,

```
class AA extends A implements Modifiable { ... }
class BB extends B implements Modifiable { ... }
```

et en définissant les méthodes données comme prototypes dans l'interface (soit la seule méthode void zoomer (int) sur l'exemple).

On peut définir des variables de type Modifiable (on ne peut pas instancier d'objet de type Modifiable) référencant des objets des classes implémentant l'interface. Une variable Modifiable peut référencer un objet AA ou un objet BB. On peut même faire un tableau de Modifiable contenant des objets implémentant l'interface Modifiable (AA ou BB). Une variable de type Modifiable ne donne accès qu'aux méthodes de l'interface. Pour accéder à une des méthodes d'un objet AA à partir d'une variable de type Modifiable, il faut forcer le type (cast) en AA. Si le transtypage échoue (l'objet n'est pas un AA), il y a lancement d'une exception de type ClassCastException.

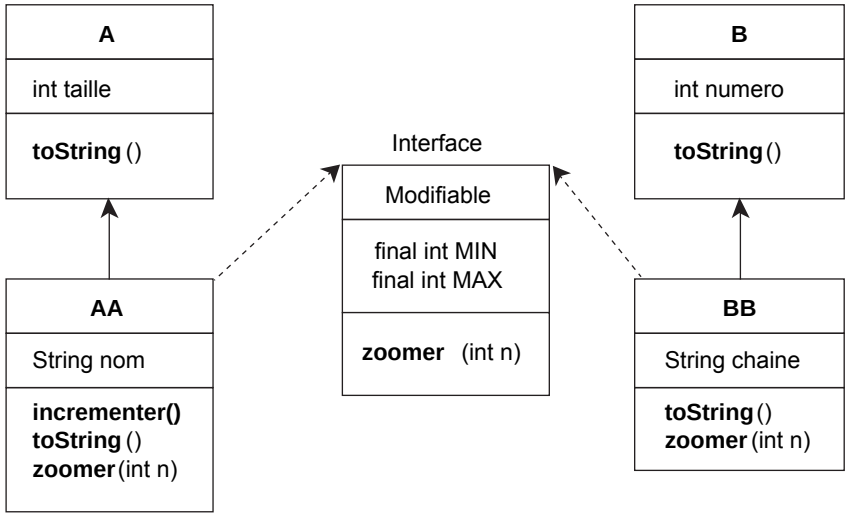

Figure 3.4 — Héritage et interface.

Le programme Java suivant permet de définir des variables de type Modifiable référençant des AA ou des BB, et de déclencher la méthode zoomer() pour des Modifiable.

```
// PPInterface.java  Programme Principal Interface

class A {
  int taille;

  A (int taille) {
    this.taille = taille;
  }

  // String des caractéristiques de A
  public String toString() {
    return "A : " + taille;
  }

}  // A

// AA hérite de A
// et implémente l'interface Modifiable
class AA extends A implements Modifiable {
```

```
  String nom;

  AA (String nom, int taille) {
    super (taille);
    this.nom = nom;
  }

  void incrementer () {
    taille++;
  }

  // String des caractéristiques de AA
  public String toString() {
    return "AA : " + super.toString() + ", AA : " + nom;
  }

  // méthode de l'interface Modifiable redéfinie pour AA
  public void zoomer(int n) {
    if (n < MIN) n = MIN;
    if (n > MAX) n = MAX;
    if (n ==0)  n = 1;
    // n > 0, on multiplie par n
    // n < 0, on divise par |n|
    taille = n > 0 ? taille*n : (int)(taille / (double) -n);
  }

}  // AA


class B {
  int valeur;

  B (int valeur) {
    this.valeur = valeur;
  }

  // String des caractéristiques de B
  public String toString() {
    return "B : " + valeur;
  }

}  // B

// BB hérite de B
// et implémente l'interface Modifiable
class BB extends B implements Modifiable {
  String chaine;

  BB (String chaine, int valeur) {
    super (valeur);
    this.chaine = chaine;
  }

  // String des caractéristiques de BB
  public String toString() {
    return "BB : " + super.toString() + ", BB : " + chaine;
  }
```

```
  // méthode de l'interface Modifiable redéfinie pour BB
  public void zoomer(int n) {
    if (n < MIN) n = MIN;
    if (n > MAX) n = MAX;
    if (n ==0)  n = 1;
    // n > 0, on multiplie par 2*n
    // n < 0, on divise par 2*|n|
    valeur = n > 0 ? valeur*2*n : (int)(valeur / (double) (-2*n));
  }

}  // BB

interface Modifiable {
  static final int MIN = -5;
  static final int MAX = +5;

  void zoomer(int n);
}  // Modifiable

public class PPInterface {                Programme Principal Interface
  public static void main (String[] args) {

    // un tableau de Modifiable contenant des AA ou des BB
    Modifiable[] tabMod = {
      new AA ("b1", 10),
      new AA ("b2", 20),
      new BB ("b3", 30),
      new BB ("b3", 50)
    };

    // on déclenche la méthode zoomer() pour les éléments du tableau
    // les AA sont divisés par 2
    // les BB sont divisés par 4
    for (int i=0; i < tabMod.length; i++) {
      tabMod[i].zoomer (-2);  // -2 : réduction
    }

    for (int i=0; i < tabMod.length; i++) {
      //System.out.println (tabMod[i].toString());
      System.out.println (tabMod[i]);
    }

    // erreur : un Modifiable ne peut pas accéder à incrementer()
    for (int i=0; i < tabMod.length; i++) {
      // erreur : incrementer n'est pas une méthode
      // de la classe Modifiable
      //tabMod[i].incrementer();
      System.out.println (tabMod[i].getClass().getName());
    }

    Modifiable m1 = tabMod[0];
    ((AA)m1).incrementer();         // il faut effectuer un transtypage
    System.out.println ("m1 : " + m1);
```

```
        Modifiable m2 = tabMod[2];                    // m2 est de type BB
        //((AA)m2).incrementer();           // Exception ClassCastException
        System.out.println ("m2 : " + m2);

    }  // main

}  // PPInterface
```

Les résultats sont les suivants :

```
                                        zoomer(-2) a modifié les valeurs
AA : A : 5, AA : b1                                                    /2
AA : A : 10, AA : b2                                                   /2
BB : B : 7, BB : b3                                                    /4
BB : B : 12, BB : b3                                                   /4
AA                  getClass().getName() fournit le nom de la classe
AA
BB
BB
m1 : AA : A : 6, AA : b1                // m1 incrémentée (de 5 à 6)
m2 : BB : B : 7, BB : b3
```

3.6 CONCLUSION

La notion d'héritage est une notion importante en programmation objet. Cette notion permet de spécialiser une classe sans avoir à tout réécrire, en utilisant sans modification, les méthodes de la super-classe si elles conviennent ou en réécrivant les méthodes qui ne conviennent plus sachant qu'une partie du traitement peut souvent être faite par les méthodes de la super-classe.

Les interfaces graphiques en Java (chapitre 5) sont organisées sous forme d'un arbre d'héritage. Tous les composants (boutons, fenêtres, zones de saisie, etc.) ont des caractéristiques communes (dimension, couleur du fond, couleur du texte, etc.). Chaque composant doit définir ses spécificités en terme d'attributs et de méthodes.

Le polymorphisme permet de définir dans la super-classe des méthodes tenant compte des spécificités des classes dérivées. En cas de redéfinition de méthodes, c'est la méthode de la classe dérivée de l'objet qui est prise en compte, et non celle de la super-classe. Les interfaces permettent de regrouper des classes ayant une ou plusieurs propriétés communes et de déclencher des traitements pour les objets des classes implémentant cette interface. Une classe ne peut hériter que d'une seule super-classe mais peut implémenter plusieurs interfaces.