Chapitre 4

Héritage

Dans certaines applications, les classes utilisées ont en commun certaines variables, méthodes de traitement ou même des signatures de méthode. Avec un langage de programmation orienté-objet, on peut définir une classe à différent niveaux d'abstraction permettant ainsi de factoriser certains attributs communs à plusieurs classes. Une classe générale définit alors un ensemble d'attributs qui sont partagés par d'autres classes, dont on dira qu'elles *héritent* de cette classe générale.

Par exemple, les classes `Carre` et `Rectangle` peuvent partager une méthode `surface()` renvoyant le résultat du calcul de la surface de la figure. Plutôt que d'écrire deux fois cette méthode, on peut définir une relation d'héritage entre les classes `Carre` et `Rectangle`. Dans ce cas, seule la classe `Rectangle` contient le code de la méthode `surface()` mais celle-ci est également utilisable sur les objets de la classe `Carre` si elle hérite de `Rectangle`.

4.1 Principe de l'héritage

L'idée principale de l'héritage est d'organiser les classes de manière hiérarchique. La relation d'héritage est unidirectionnelle et, si une classe B hérite d'une classe A, on dira que B est une sous-classe de A. Cette notion de sous-classe signifie que la classe B est un cas particulier de la classe A et donc que les objets instanciant la classe B instancient également la classe A.

Prenons comme exemple des classes `Carre`, `Rectangle` et `Cercle`. La figure 4.1 propose une organisation hiérarchique de ces classes telle que `Carre` hérite de `Rectangle` qui hérite, ainsi que `Cercle`, d'une classe `Forme`.

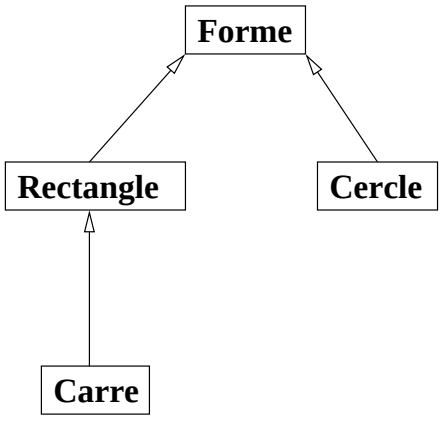

FIGURE 4.1 – Exemple de relations d'héritage

Pour le moment, nous considérerons la classe `Forme` comme vide (c'est-à-dire sans aucune variable ni méthode) et nous nous intéressons plus particulièrement aux classes `Rectangle` et `Carre`.

La classe `Rectangle` héritant d'une classe vide, elle ne peut profiter d'aucun de ses attributs et doit définir toutes ses variables et méthodes. Une relation d'héritage se définit en Java par le mot-clé `extends` utilisé comme dans l'exemple suivant :

```
public class Rectangle extends Forme {

   private int largeur;
   private int longueur;

   public Rectangle(int x, int y) {
      this.largeur = x;
      this.longueur = y;
   }

   public int getLargeur() {
      return this.largeur;
   }

   public int getLongueur() {
      return this.longueur;
   }

   public int surface() {
      return this.longueur * this.largeur;
   }

   public void affiche() {
      System.out.println("rectangle " + longueur + "x" + largeur);
   }
}
```

En revanche, la classe `Carre` peut bénéficier de la classe `Rectangle` et ne nécessite pas la réécriture de ces méthodes si celles-ci conviennent à la sous-classe. Toutes les méthodes et variables de la classe `Rectangle` ne sont néanmoins pas accessibles dans la classe `Carre`. Pour qu'un attribut puisse être utilisé dans une sous-classe, il faut que son type d'accès soit `public` ou `protected`, ou, si les deux classes sont situées dans le même package, qu'il utilise le type d'accès par défaut. Dans cet exemple, les variables `longueur` et `largeur` ne sont pas accessibles dans la class `Carre` qui doit passer par les méthodes `getLargeur()` et `getLongueur()`, déclarées comme publiques.

4.1.1 Redéfinition

L'héritage intégral des attributs de la classe `Rectangle` pose deux problèmes :

1. il faut que chaque carré ait une longueur et une largeur égales ;
2. la méthode `affiche` écrit le mot "`rectangle`" en début de chaîne. Il serait souhaitable que ce soit "`carré`" qui s'affiche.

De plus, les constructeurs ne sont pas hérités par une sous-classe. Il faut donc écrire un constructeur spécifique pour `Carre`. Ceci nous permettra de résoudre le premier problème en écrivant un constructeur qui ne prend qu'un paramètre qui sera affecté à la longueur et à la largeur. Pour attribuer une valeur à ces variables (qui sont privées), le constructeur de `Carre` doit faire appel au

constructeur de `Rectangle` en utilisant le mot-clé `super` qui fait appel au constructeur de la classe supérieure comme suit :

```
public Carre(int cote) {
  super(cote,cote);
}
```

☛ **Remarques :**

- L'appel au constructeur d'une classe supérieure doit toujours se situer dans un constructeur et toujours en tant que première instruction ;
- Si aucun appel à un constructeur d'une classe supérieure n'est fait, le constructeur fait appel implicitement à un constructeur vide de la classe supérieure (comme si la ligne `super()` était présente). Si aucun constructeur vide n'est accessible dans la classe supérieure, une erreur se produit lors de la compilation.

Le second problème peut être résolu par une *redéfinition* de méthode. On dit qu'une méthode d'une sous-classe redéfinit une méthode de sa classe supérieure, si elles ont la même signature mais que le traitement effectué est ré-écrit dans la sous-classe. Voici le code de la classe `Carre` où sont résolus les deux problèmes soulevés :

```
public class Carre extends Rectangle {

   public Carre(int cote) {
      super(cote, cote);
   }

   public void affiche() {
      System.out.println("carré " + this.getLongueur());
   }
}
```

Lors de la redéfinition d'une méthode, il est encore possible d'accéder à la méthode redéfinie dans la classe supérieure. Cet accès utilise également le mot-clé `super` comme préfixe à la méthode. Dans notre cas, il faudrait écrire `super.affiche()` pour effectuer le traitement de la méthode `affiche()` de `Rectangle`.

Enfin, il est possible d'interdire la redéfinition d'une méthode ou d'une variable en introduisant le mot-clé `final` au début d'une signature de méthode ou d'une déclaration de variable. Il est aussi possible d'interdire l'héritage d'une classe en utilisant `final` au début de la déclaration d'une classe (avant le mot-clé `class`).

4.1.2 Polymorphisme

Le polymorphisme est la faculté attribuée à un objet d'être une instance de plusieurs classes. Il a une seule classe "réelle" qui est celle dont le constructeur a été appelé en premier (c'est-à-dire la classe figurant après le `new`) mais il peut aussi être déclaré avec une classe supérieure à sa classe réelle. Cette propriété est très utile pour la création d'ensembles regroupant des objets de classes différentes comme dans l'exemple suivant :

```
Forme[] tableau = new Forme[4];
tableau[0] = new Rectangle(10,20);
tableau[1] = new Cercle(15);
tableau[2] = new Rectangle(5,30);
tableau[3] = new Carre(10);
```

L'opérateur `instanceof` peut être utilisé pour tester l'appartenance à une classe comme suit :

```
for (int i = 0 ; i < tableau.length ; i++) {
   if (tableau[i] instanceof Forme)
      System.out.println("element " + i + " est une forme");
   if (tableau[i] instanceof Cercle)
      System.out.println("element " + i + " est un cercle");
   if (tableau[i] instanceof Rectangle)
      System.out.println("element " + i + " est un rectangle");
   if (tableau[i] instanceof Carre)
      System.out.println("element " + i + " est un carré");
}
```

L'exécution de ce code sur le tableau précédent affiche le texte suivant :

```
element[0] est une forme
element[0] est un rectangle
element[1] est une forme
element[1] est un cercle
element[2] est une forme
element[2] est un rectangle
element[3] est une forme
element[3] est un rectangle
element[3] est un carré
```

L'ensemble des classes Java, y compris celles écrites en dehors de l'API, forme une hiérarchie avec une racine unique. Cette racine est la classe `Object` dont hérite toute autre classe. En effet, si vous ne précisez pas explicitement une relation d'héritage lors de l'écriture d'une classe, celle-ci hérite par défaut de la classe `Object`. Grâce à cette propriété, des classes génériques [1] de création et de gestion d'un ensemble, plus élaborées que les tableaux, regroupent des objets appartenant à la classe `Object` (donc de n'importe quelle classe).

Une des propriétés induites par le polymorphisme est que l'interpréteur Java est capable de trouver le traitement à effectuer lors de l'appel d'une méthode sur un objet. Ainsi, pour plusieurs objets déclarés sous la même classe (mais n'ayant pas la même classe réelle), le traitement associé à une méthode donné peut être différent. Si cette méthode est redéfinie par la classe réelle d'un objet (ou par une classe située entre la classe réelle et la classe de déclaration), le traitement effectué est celui défini dans la classe la plus spécifique de l'objet et qui redéfinie la méthode.

Dans notre exemple, la méthode `affiche()` est redéfinie dans toutes les sous-classes de `Forme` et les traitements effectués sont :

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

1. voir par exemple les classes `java.util.Vector`, `java.util.Hashtable`, …

Résultat :

```
rectangle 10x20
cercle 15
rectangle 5x30
carré 10
```

Dans l'état actuel de nos classes, ce code ne pourra cependant pas être compilé. En effet, la fonction `affiche()` est appelée sur des objets dont la classe déclarée est `Forme` mais celle-ci ne contient aucune fonction appelée `affiche()` (elle est seulement définie dans ses sous-classes). Pour compiler ce programme, il faut transformer la classe `Forme` en une interface ou une classe abstraite tel que cela est fait dans les sections suivantes.

4.2 Interfaces

Une interface est un type, au même titre qu'une classe, mais abstrait et qui donc ne peut être instancié (par appel à `new` plus constructeur). Une interface décrit un ensemble de signatures de méthodes, sans implémentation, qui doivent être implémentées dans toutes les classes qui *implémentent* l'interface. L'utilité du concept d'interface réside dans le regroupement de plusieurs classes, tel que chacune implémente un ensemble commun de méthodes, sous un même type. Une interface possède les caractéristiques suivantes :

- elle contient des signatures de méthodes ;
- elle ne peut pas contenir de variables ;
- une interface peut hériter d'une autre interface (avec le mot-clé `extends`) ;
- une classe (abstraite ou non) peut implémenter plusieurs interfaces. La liste des interfaces implémentées doit alors figurer après le mot-clé `implements` placé dans la déclaration de classe, en séparant chaque interface par une virgule.

Dans notre exemple, `Forme` peut être une interface décrivant les méthodes qui doivent être implémentées par les classes `Rectangle` et `Cercle`, ainsi que par la classe `Carre` (même si celle-ci peut profiter de son héritage de `Rectangle`). L'interface `Forme` s'écrit alors de la manière suivante :

```
public interface Forme {
   public int surface() ;
   public void affiche() ;
}
```

Pour obliger les classes `Rectangle`, `Cercle` et `Carre` à implémenter les méthodes `surface()` et `affiche()`, il faut modifier l'héritage de ce qui était la classe `Forme` en une implémentation de l'interface définie ci-dessus :

```
public class Rectangle implements Forme {
   ...
}
```

et

```
public class Cercle implements Forme {
   ...
}
```

Cette structure de classes nous permet désormais de pouvoir compiler l'exemple donné dans la section 4.1.2 traitant du polymorphisme. En déclarant un tableau constitué d'objets implémentant l'interface `Forme`, on peut appeler la méthode `affiche()` qui existe et est implémentée par chaque objet.

Si une classe implémente une interface mais que le programmeur n'a pas écrit l'implémentation de toutes les méthodes de l'interface, une erreur de compilation se produira sauf si la classe est une classe abstraite.

4.3 Classes abstraites

Le concept de classe abstraite se situe entre celui de classe et celui d'interface. C'est une classe qu'on ne peut pas directement instancier car certaines de ses méthodes ne sont pas implémentées. Une classe abstraite peut donc contenir des variables, des méthodes implémentées et des signatures de méthode à implémenter. Une classe abstraite peut implémenter (partiellement ou totalement) des interfaces et peut hériter d'une classe ou d'une classe abstraite.

Le mot-clé `abstract` est utilisé devant le mot-clé `class` pour déclarer une classe abstraite, ainsi que pour la déclaration de signatures de méthodes à implémenter.

Imaginons que l'on souhaite attribuer deux variables, `origine_x` et `origine_y`, à tout objet représentant une forme. Comme une interface ne peut contenir de variables, il faut transformer `Forme` en classe abstraite comme suit :

```
public abstract class Forme {
   private int origine_x;
   private int origine_y;

   public Forme() {
      this.origine_x = 0;
      this.origine_y = 0;
   }

   public int getOrigineX() {
      return this.origine_x;
   }

   public int getOrigineY() {
      return this.origine_y;
   }

   public void setOrigineX(int x) {
      this.origine_x = x;
   }

   public void setOrigineY(int y) {
      this.origine_y = y;
   }

   public abstract int surface();

   public abstract void affiche();
}
```

De plus, il faut rétablir l'héritage des classes `Rectangle` et `Cercle` vers `Forme` :

```
public class Rectangle extends Forme {
    ...
}
```

et

```
public class Cercle extends Forme {
    ...
}
```

Lorsqu'une classe hérite d'une classe abstraite, elle doit :

- soit implémenter les méthodes abstraites de sa super-classe en les dotant d'un corps ;
- soit être elle-même abstraite si au moins une des méthodes abstraites de sa super-classe reste abstraite.

4.4 Classes et méthodes génériques

Il est parfois utile de définir des classes paramétrées par un type de données (ou une classe). Par exemple, dans le package `java.util`, de nombreuses classes sont génériques et notamment les classes représentant des ensembles (`Vector`, `ArrayList`, etc.). Ces classes sont génériques dans le sens où elles prennent en paramètre un type (classe ou interface) quelconque `E`. `E` est en quelque sorte une variable qui peut prendre comme valeur un type de donné. Ceci se note comme suit, en prenant l'exempe de `java.util.ArrayList` :

```
package java.util ;

public class ArrayList<E>
      extends AbstractList<E>
      implements List<E>, ...
{
   ...
   public E set(int index, E element) {
      ...
   }

   public boolean add(E e) {
      ...
   }
   ...
}
```

Nous pouvons remarquer que le type passé en paramètre est noté entre chevrons (ex : `<E>`), et qu'il peut ensuite être réutilisé dans le corps de la classe, par des méthodes (ex : la méthode `set` renvoie un élément de classe `E`).

Il est possible de définir des contraintes sur le type passé en paramètre, comme par exemple une contrainte de type `extends` [2] :

2. Ici, on utilise `T extends E` pour signaler que le type `T` est un sous type de `E`, que `E` soit une classe ou une interface (on n'utilise pas `implements`).

```
public class SortedList<T extends Comparable<T>> {
    ...
}
```

Ceci signifie que la classe `SortedList` (liste ordonnée que nous voulons définir) est paramétrée par le type `T` qui doit être un type dérivé (par héritage ou interfaçage) de `Comparable<T>`. En bref, nous définissons une liste ordonnée d'éléments comparables entre eux (pour pouvoir les trier), grâce à la méthode `int compareTo(T o)` de l'interface `Comparable`[3] qui permet de comparer un `Comparable` à un élément de type `T`.

3. Voir `java.lang.Comparable` pour plus d'information sur cette interface.