Chapitre 11

Polymorphisme

Comme le terme le suggère, le polymorphisme désigne la capacité d'un objet à pouvoir prendre plusieurs formes (au sens de *types*) différentes. Il s'agit d'une caractéristique fondamentale des langages objets, liée à l'héritage.

Ce mécanisme repose sur la faculté d'un objet d'une classe dérivée à pouvoir être considéré soit comme une instance de sa propre classe, soit comme une instance de sa super-classe.

1 Transtypage ascendant et transtypage descendant

1.1 Inclusion de types et transtypage

Il semble naturel qu'une valeur entière puisse être affectée à une variable de type réel : il n'y a pas de risque de perte de précision, et en maths $\mathbb{N} \subset \mathbb{R}$. En revanche, l'inverse n'est pas vrai. Pour pouvoir affecter une variable réelle à une variable entière, il faut tronquer la partie décimale du nombre réel.

L'affectation d'une variable d'un type donné à une variable d'un autre type s'appelle un *transtypage* (ou *cast* en anglais).

Soient n une variable entière (type Java int) et x une variable réelle (type Java float).

On peut considérer que le type `int` est *inclus* dans le type `float`. Ce transtypage n'a pas à être déclaré explicitement :

```
x = n;
```

est valide en Java.

Par contre, le type `float` *n'est pas inclus* dans le type `int`, et transtyper une variable réelle en une variable entière nécessite un transtypage explicite pour être accepté à la compilation :

```
n = (int) x;
```

Cette notion d'inclusion de types s'applique également aux objets définis par héritage. En effet, une instance d'une sous-classe est aussi une instance de sa super-classe, et en ce sens on peut dire que le type défini par une sous-classe est inclus dans celui défini par sa super-classe.

Un transtypage *ascendant* consiste à voir un objet d'une sous-classe comme un objet de sa super-classe. Le transtypage ascendant est toujours valide et autorisé, et ne nécessite pas d'être déclaré explicitement.

L'opération inverse, consistant à transformer un objet d'une classe donnée en un objet d'une sous-classe, est appelée un transtypage *descendant*. Un transtypage descendant doit toujours être déclaré explicitement pour être accepté à la compilation. Par contre, la validité du transtypage ne pourra être contrôlée qu'à l'exécution. Pour que le transtypage descendant soit valide, il faut que l'objet qu'on cherche à transtyper soit effectivement en réalité une instance du type souhaité. Sinon, une erreur (exception) est déclenchée à l'exécution.

Exemples

```
public class ACast {
}

public class BCast extends ACast {
    public static void main(String[] args) {
        ACast a1 = new ACast(), a2 = new ACast();
        BCast b1 = new BCast(), b2 = new BCast(), b3 = new BCast();

        a1 = b1; // OK compil (sans cast explicite), OK exec
        b2 = a2; // erreur compilation (il faut caster)
        b2 = (BCast) a2; // OK compil, erreur execution
        b3 = (BCast) a1; // OK compil, OK exec
    }
}
```

Les horaires

```
public class CastHoraires {
    public static void main(String[] args) {
        Horaire h1 = new Horaire(), h2 = new Horaire();
        HorairePrecis hp1 = new HorairePrecis(), hp2 = new HorairePrecis();

        h1 = hp1; // OK compil, OK exec
        hp2 = h2; // Erreur compil
        hp2 = (HorairePrecis) h2; // OK compil, erreur exec
        hp1 = (HorairePrecis) h1; // OK compil, OK exec
    }
}
```

1.2 Le mécanisme de liaison dynamique, ou *late binding*

Exemple

```
public class Dumb {
    public void whoAmI() {
      System.out.println("I am dumb") ;
    }
}

public class Dumber extends Dumb {
    public void whoAmI() {
      System.out.println("I am dumber") ;
    }
}

public class GoDumb {
    public static void main(String[] args) {
      Dumb dumb ;
      Dumber dumber = new Dumber() ;
      dumb = dumber ; // transtypage de Dumber en Dumb
      dumb.whoAmI() ; // affiche "I am dumber" ! ! !
    }
}
```

Dumber hérite de Dumb. La méthode whoAmI(), définie dans Dumb, est *redéfinie* dans Dumber. dumb est une variable de type Dumb et dumber est une instance de Dumber.

Quand on réalise le transtypage (ascendant) suivant : dumb = dumber, du point de vue typage, dumber est vu comme un Dumb. Pourtant, quand on invoque whoAmI() sur dumb, cela affiche « I am dumber » ! dumber n'a pas complètement perdu son identité lors du transtypage. Il existe à la fois sous la forme Dumb et Dumber.

En résumé, dumber est *syntaxiquement* un Dumb, mais *dynamiquement* (c'est à dire à l'exécution) un Dumber.

Ce comportement peut sembler curieux (voire déroutant) au premier abord ! Il est cependant d'une importance cruciale qu'il en soit ainsi.

Cela permet de stocker dans une même structure des objets de types variés, mais descendant tous d'un même type de base. On les stocke en tant qu'instances de la classe de base (ce qui est toujours autorisé), dans une structure de tableau par exemple. Le code est syntaxiquement correct et peut donc être compilé. Cependant, au moment de manipuler

ces objets, on souhaitera très certainement que le comportement invoqué soit bien celui de l'objet réel, bien que celui-ci ait été transtypé !

Au moment *d'exécuter* une méthode sur une instance, l'interpréteur java recherche la « version » de la méthode qu'il doit appliquer. Il la recherche dans l'instance elle-même. Si la méthode n'y est pas définie, il la recherche dans l'instance interne de la super-classe, puis éventuellement dans l'instance de la « super super classe », etc. La recherche s'arrête à la première super-classe définissant la méthode. On note qu'elle est garantie d'exister quelque part car cela a déjà été vérifié (syntaxiquement) par la compilation.