IFT1025 été 2010

Miklós Csűrös

24 mai 2010

2 Héritage

2.1 Sous-classes

En vérité chaque classe est la sous-classe d'une autre classe, sauf `Object` qui est l'objet de racine. On utilise l'extension d'une classe pour changer l'implémentation.

```
public class Rectangle extends Shape
{
    public double aire()
    {
        // code spécifique à Rectangle
    }
}
```

La conversion est automatique d'une sous-classe vers la superclasse mais non pas à l'inverse.

```
    Shape S = new Rectangle(); // OK, Rectangle est la sous-classe de Shape
    Object O = S; // OK, Shape est une sous-classe de Object
    Rectangle R1 = S; // ERREUR: pas de conversion automatique
    Rectangle R2 = (Rectangle) S; // OK: conversion explicite possible
```

2.2 Règles de héritage

Les méthodes et les variables de la superclasse sont **héritées** dans la sous-classe si elles sont visibles : celles avec accès `protected` et `public` sont héritées, celles avec accès par défaut sont héritées seulement si la sous-classe appartient au même package, et celles avec accès `private` ne sont pas héritées.

Les constructeurs ne sont jamais hérités (mais le constructeur sans arguments est là toujours).

La sous-classe peut définir de nouveaux constructeurs, méthodes et variables.

2.3 *Hiding* et *overriding*

Une méthode d'objet peut être **redéfinie**[1] dans la sous-classe par une méthode de la même signature : il s'agit de l'*overriding*. Si on définit une méthode statique avec la même signature, ou une variable (de classe ou de l'objet) dans la sous-classe, la version de la superclasse devient **cachée** : il s'agit de *hiding*.

Il n'est pas permis de restreindre l'accès lors de redéfinition dans la sous-classe (p.e. de `public` à `private`), mais on peut l'élargir (p.e. de niveau package à `protected`). Il est aussi permis d'utiliser une sous-classe comme valeur retournée (p.e., la méthode dans la superclasse retourne `Object` mais dans la sous-classe retourne `Integer`).

La sous-classe peut accéder aux méthodes et variables de la superclasse par le mot clé `super`.

[1] sauf pour les méthodes `final`

```
public class Panneau extends JPanel
{
    public int var; // variable pur la sous-classe

    @Override // mot-clé optionnel en Java 6
    public void paintComponent(Graphics g) // overriding
    {
     super.paintComponent(g); // dessin de l'arrière-plan
     ... code spécifique à Panneau
    }

    public StringBuffer toString() // ERREUR: overriding Object.toString()
    {
       ...
    }
}
class Panneau2 extends Panneau
{
    public int var; // hiding Panneau.var
}
```

2.4 Constructeurs

Les constructeurs ne sont pas hérités. Par contre, tout objet possède un constructeur sans argument par défaut, sauf si d'autres constructeurs sont définis. Le constructeur de la superclasse peut être appellé par `super`. Par défaut, la première instruction du constructeur de la sous-classe est un appel implicite à `super()`, sauf si un autre constructeur de la superclasse est appellé explicitement.

```
public class Panneau extends JPanel
{
    public Panneau(int x)
    {
        // appel implicite super()
        ... code spécifique à Panneau
    }

    public Panneau(double m)
    {
        super(false); // appel du constructeur JPanel(boolean)
        ... code
    }
}
class Panneau2 extends Panneau
{
    // ERREUR: le constructeur par défaut Panneau2()
    // essaie d'appeler super() mais il n'existe pas
    // de constructeur sans argument en Panneau
}
```

2.5 Polymorphisme

Si on a des noms/signatures identiques dans la sous-classe et la superclasse, comment Java choisit-il la version correcte ? Dans le cas des méthodes d'objet, c'est au temps de l'exécution que la version est choisie (liason dynamique — *late binding*). Dans le cas de méthodes de la classe, et les variables d'objet ou de la classe, c'est au temps de compilation que la version est choisie (liaison statique — *early binding*). Donc : possibilité de *overriding* ↔ liaison dynamique, et *hiding* ↔ liaison statique.

```
    Object X = new Integer(5); // OK, conversion vers superclasse
    System.out.println(X.toString()); // c'est Integer.ToString() qui est executé
```

```
class Cronos
{
    public static String nomC() {return "Cronos";}
    public String nomO {return "Cronos";}
}
class Ouranos extends Cronos
{
    public static String nomC() {return "Ouranos";} // hiding
    public String nomO {return "Ouranos";} // overriding
    public static void main(String[] args)
    {
        Cronos x = new Ouranos(); // OK, conversion vers superclasse
        System.out.println(x.nomO()); // "Ouranos", late binding
        System.out.println(x.nomC()); // "Cronos", early binding
    }
}
```

2.6 Classe abstraite

Une classe abstraite est déclarée par le mot-clé `abstract`. Dans une telle classe, on peut déclarer des méthodes abstraites sans définition dont le code doit être fourni dans les sous-classes non-abstraites.

Il n'est pas permis de déclarer une méthode abstraite si elle ne peut pas être redéfinie : les combinaisons `final+abstract`, `private+abstract`, `static+abstract` mènent à des erreurs de compilation.

```
public abstract class Liste
{
    public abstract Object get(int idx); // définition dans une sous-classe
    public abstract int size(); // définition dans une sous-classe
    public Object toArray()
    {
        Object[] retval = new Object[size()];
        for (int i=0; i<size(); i++)
        {
            retval[i] = get(i);
        }
        return retval;
    }
}
class ArrayList extends Liste
{
    @Override
    public Object get(int idx)
    {
        ... code
    }

    @Override
    public int size()
    {
        ... code
    }
}
```

2.7 Mot-clé final

Le mot-clé `final` sert à interdire l'extension d'une classe, la redéfinition d'une méthode, ou la modification de valeur d'une variable. Exemples :

- ⋆ `public final class Zeus` — pas de sous-classes
- ⋆ `public final Object[] toArray()` — pas de redéfinition
- ⋆ `final int z=5;` — pas d'affectation
- ⋆ `public static final int TAILLE=3;` — pas d'affectation (constante de la classe)