L3 MIASHS

INF F5 — Héritage

UGA UFR SHS Université Grenoble Alpes

Daniel.Bardou@univ-grenoble-alpes.fr

Sommaire

- Définitions
- Constructeurs et héritage
- Pseudo-variables
- Ajout d'attributs
- Ajouts et redéfinitions de méthodes
- Polymorphisme d'inclusion
- Modificateur **final**
- Modificateur **abstract**

Définitions

Héritage

- L'héritage est un mécanisme fondamental des langages de programmation par objets.
- Il permet d'obtenir, à partir de descriptions d'objets déjà existantes, de nouvelles descriptions d'objets plus spécifiques.
- En Java, l'héritage est spécifié entre classes et ce sont les propriétés d'instance (attributs et méthodes) qui sont héritées d'une classe à l'autre.

Super-classe et sous-classe

- Si la classe B hérite de la classe A, alors on dit que
  - A est super-classe de B,
  - B est une sous-classe de A.
- B hérite de A :
  - les attributs (d'instance et de classe),
  - les méthodes (d'instance et de classe).
- B n'hérite pas de A :
  - les constructeurs,
  - les blocs **static**.

Syntaxe

- En Java, l'héritage est spécifié en utilisant le mot réservé extends.
- Exemple :

```
public class B extends A {
        // la classe B hérite de la classe A
        // ...
    }
```

Héritage simple

- En Java, l'héritage est simple : toute classe admet **une et une seule** super-classe directe.
- La définition :

```
public class A {
    // ...
}
```

- équivaut à :

```
public class A extends Object {
    // ...
}
```

La classe Object

- Object est la racine de la hiérarchie d'héritage en Java : toute classe hérite (directement ou indirectement) de Object.
- Object détient des définitions de méthodes exécutables sur tout objet Java, notamment :
  - **public boolean** equals(Object obj)
  - **public** String toString()
  - **public** Class getClass()

Modificateur protected

- Dans une classe, le modificateur de visibilité **protected** permet de rendre visible un attribut ou une méthode depuis les sous-classes.
- **protected** est une visibilité plus ouverte que l'accès « paquetage » : les sous-classes peuvent être définies dans des paquetages différents.
- La bonne pratique reste d'utiliser **private** au maximum pour les attributs.

Héritage et constructeurs

Ordre d'exécution des constructeurs

- À l'instanciation d'une classe, l'initialisation de la partie héritée de l'instance doit être effectuée avant celle de la partie localement définie.
- Un constructeur de la super-classe doit donc toujours être exécuté avant l'exécution d'un constructeur de la classe.
- Les constructeurs sont donc exécutés dans l'ordre des classes les plus générales aux plus spécifiques.

Exemple

```
public class Point {
    private double x, y;

    public Point() {
        this(0, 0);
    }
    public Point(double abs, double ord) {
        x = abs;
        y = ord;
    }
}
```

Exemple

```
public class Point3D extends Point {
    private double z;

    public Point3D() {
        this(0, 0, 0);
    }

    public Point3D(double x, double y, double z) {
        super(x, y); // appel « super-constructeur »
        this.z = z;
    }
}
```

Première instruction de constructeur

- La première instruction d'un constructeur est toujours l'appel d'un constructeur de la super-classe en utilisant la syntaxe **super**(...), ou l'appel d'un autre constructeur de la classe en utilisant la syntaxe **this**(...).
- Si rien n'est indiqué, le compilateur Java considère par défaut un appel au constructeur sans paramètre de la super-classe, soit **super**().

Exemple

```
public class A {
    protected int a;

    public A(int val) {
        a = val;
    }
}
```

```
public class B
        extends A {

    public B(int val) {
        a = val;
    }
}
```

Erreur de compilation : constructeur A() non défini !

Pseudo-variables

Pseudo-variables

- Il existe 2 pseudo-variables que l'on peut utiliser dans les méthodes d'une classe :
  - **this** référence l'instance courante,
  - **super** référence l'instance courante limitée à sa partie héritée.
- **this** et **super** sont appelées pseudo-variables car :
  - leur valeur change selon le contexte,
  - on ne peut pas modifier leur valeur explicitement.

Ajout d'attributs

Ajout d'attributs

- Il est fréquent de définir de nouveaux attributs dans une sous-classe qui viennent compléter ceux hérités de la super-classe.
- Par exemple, on a défini un attribut `z` dans la classe `Point3D` pour compléter les attributs `x` et `y` définis dans la classe `Point` et hérités par `Point3D`.
- Chaque instance de la sous-classe détient sa propre valeur pour chaque attribut localement défini et chaque attribut hérité.

Ajout d'attributs homonymes

- Il est possible, **mais fortement déconseillé** (ce n'est souvent fait que par erreur), de définir dans une classe des attributs de même nom que ceux hérités.
- (Mauvais) Exemple :

```
public class Point3D extends Point {
    private double x;
    // pas d'erreur de compilation !
}
```

Ajout et redéfinitions de méthodes

Ajout et rédéfinitions de méthodes

- Il est possible de définir de nouvelles méthodes dans une sous-classe.
- Il est également possible de donner une nouvelle définition d'une méthode héritée.
- Il n'est pas possible de changer la signature d'une méthode héritée en la redéfinissant.

Exemple

```
public class Point{
    private double x, y;

    public String toString() {
        /* redéfinition de toString()
           héritée de Object */
        return "(" + x + ", " + y + ")" ;
    }
}
```

Autre Exemple

```
public class Point3D extends Point{
    private double z;

    public String toString() {
        // redéfinition avec appel sur super
        String s = super.toString();
        int l = s.length();
        return s.substring(0, l – 1) + ", " + z + ")";
    }
}
```

Polymorphisme d'inclusion

Polymorphisme

- Polymorphisme ad-hoc : plusieurs méthodes peuvent être définies sur plusieurs classes (éventuellement sans héritage) et conduirent à des exécutions différentes.
- Polymorphisme d'inclusion : on peut utiliser une instance d'une sous-classe d'une classe C en lieu et place d'une instance de C. Toute sous-classe de C définit au moins le même comportement pour ses instances sous éventuellement une forme plus spécifique.
- **extends** peut être assimilé à « est-un »

Exemple 1

```
Point p = new Point();
Point3D p3D = new Point3D();

p = p3D; // Ok
p3D = p; // Erreur de compilation
p3D = (Point3D) p; // Ok
```

Exemple 2

```
Point p = new Point();
Point3D p3D = new Point3D();
Point[] t = new Point[2];
t[0] = p;
t[1] = p3D;
for (int i = 0; i < t.length; i ++)
    System.out.println(t[i].toString());
// affiche sur la console :
// (0, 0)
// (0, 0, 0)
```

Polymorphisme et typage

- Une référence sur un objet dans un programme a :
  - un type déclaré, c'est le type donné à la déclaration de la référence, sur lequel le compilateur fait des vérifications,
  - un type réel (ou précis), c'est le type de l'objet référencé (la classe dont il est instance directe),
  - potentiellement beaucoup d'autres types en relation d'héritage avec le type déclaré et le type réel.

Opérateur instanceof

- L'opérateur **instanceof** permet de savoir si une référence possède un type donné.
- Exemple

```
Object obj = new Point3D();
Point p = new Point();
System.out.println(obj instanceof Point); // affiche true
System.out.println(obj instanceof Object); // true
System.out.println(obj instanceof Point3D); // true
System.out.println(obj instanceof String); // false
System.out.println(p instanceof Point3D); // false
System.out.println(p instanceof String); // Erreur
```

Modificateur final

Modificateur final

- Le modificateur **final** permet de limiter l'héritage :
  - une classe **final** ne peut pas avoir de sous-classes,
  - une méthode **final** ne peut pas être redéfinie dans une sous-classe,
  - pour rappel, une variable ou un attribut **final** est une constante.

Intérêt de final

- Pour une classe (sécurité essentiellement) :
  - impossibilité de rentrer dans la sphère de visibilité protected en définissant une classe caduque,
  - impossibilité de substituer des instances d'un autre type réel.
- Pour une méthode :
  - sécurité, impossibilité de redéfinir et « dénaturer le comportement »,
  - efficacité, génération de code « inline ».

Modificateur abstract

Utilisation de l'héritage

- On peut utiliser l'héritage selon 2 axes complémentaires :
  - pour réutiliser des définitions existantes en créant de nouvelles sous-classes à des classes existantes,
  - pour factoriser du code commun à plusieurs classes en leur créant une super-classe commune et en y plaçant ce code (cela évite la redondance du code et en facilite la correction et l'évolution).
- L'ensemble des définitions factorisées dans une super-classe n'est pas toujours « suffisamment complet » pour constituer la définition d'objets « viables », on a donc introduit dans les langages de programmation par objets la notion de classe abstraite.

Classe abstraite

- Une classe abstraite est une classe pour laquelle il est interdit d'avoir des instances directes.
- Sa définition est introduite avec le modificateur **abstract**
- Les classes abstraites sont utiles
  - pour la factorisation de définitions devant être héritées par plusieurs sous-classes,
  - pour définir des « classes bibliothèques » de programmes qui ne contiennent que des méthodes de classe (la classe `Math` par exemple).

Classe abstraite

- Exemple :
  - ```
    public abstract class C {
        // ...
        public C() {
            // ...
        }
        // ...
    }
    ```
  - Erreur de compilation si **new** C() apparaît dans un programme.
    ```

Méthode abstraite

- Une méthode abstraite est une méthode pour laquelle la définition se limite à son entête (visibilité, type de retour et signature).
- Sa définition est introduite avec le modificateur **abstract**
- Détenir (localement ou par héritage) une méthode abstraite est une condition suffisante (mais non nécessaire) pour qu'une classe soit abstraite.

Exemple (1/3)

```
public abstract class Forme {
    private Point pointRef;

    public Forme(Point p) {
        pointRef = p;
    }
    public abstract double perimetre();
    public abstract double surface();

    public void deplacer(double dx, double dy) {
        pointRef.setX(pointRef.getX() + dx);
        pointRef.setY(pointRef.getY() + dy);
    }
}
```

Exemple (2/3)

```
public class Disque extends Forme {
    // pointRef hérité correspond au centre du disque
    private double rayon;

    public Disque(Point centre, double rayon) {
        super(centre);
        this.rayon = rayon;
    }
    public double perimetre() {
        return 2 * Math.PI * rayon;
    }
    public double surface() {
        return Math.PI * rayon * rayon;
    }
}
```

Exemple (3/3)

```
public class Rectangle extends Forme {
    // pointRef hérité correspond au coin supérieur gauche
    private double hauteur, largeur;

    public Rectangle(Point coin, double hauteur, double largeur) {
        super(coin);
        this.hauteur = hauteur;
        this.largeur = largeur;
    }
    public double perimetre() {
        return 2 * (hauteur + largeur);
    }
    public double surface() {
        return hauteur * largeur;
    }
}
```