

Algorithmique et programmation par objets

Inf F3

Licence 2 MIASHS

Université Grenoble Alpes

Jerome.David@univ-grenoble-alpes.fr

2024-2025

<http://miashs-www.u-ga.fr/~davidjer/inff3/>

Cours 6 - La réutilisation des classes

- Composition, délégation et héritage
- Surcharge, redéfinition
- Transtypage ascendant

Rappels

- Une classe décrit une famille d'objets qui ont commun :
 - Les mêmes types de données : les attributs
 - Les mêmes opérations (ou comportements) : les méthodes
 - Les mêmes « moules » pour la création et l'initialisation d'instances : les constructeurs
- L'ensemble des attributs, méthodes et constructeurs sont appelés membres de la classe

Le but du cours d'aujourd'hui :
Comment peut-on réutiliser le code d'une classe pour servir de brique dans une autre classe ?

Réutilisation des classes

- Java offre deux moyens pour réutiliser le code de classes existantes
 - La composition
 - On crée des instances de classes existantes dans des nouvelles classes
 - L'héritage
 - On crée des classes qui reprennent le type d'une classe existante et on ajoute des fonctionnalités (sans modifier) le code existant

La composition

- Cela consiste à placer une référence vers un objet dans une classe
 - Nous l'avons déjà utilisé plusieurs fois
 - Voir les TP2 et 3

```
public class Personne {  
    private String nom;  
    private String prenom;  
  
    public Personne(String n, String p) {  
        nom=n;  
        prenom=p;  
    }  
  
    public String toString() {  
        return "Je m'appelle "+prenom+" "+nom;  
    }  
}
```

La classe Personne est composée de deux chaînes de caractères.

Dans la méthode toString, on réutilise (implicitement) la méthode concatenate sur les chaînes de caractères

La composition autre exemple

- Une fenêtre dans laquelle on peut ajouter des boutons

```
import javax.swing.JButton;
import javax.swing.JFrame;

public class UneFenetre {
    private JFrame fenetre;

    public UneFenetre() {
        fenetre=new JFrame();
        fenetre.setVisible(true);
    }

    public void ajouterUnBouton(String texte) {
        JButton unBouton = new JButton(texte);
        fenetre.getContentPane().add(unBouton);
        fenetre.pack();
    }
}
```

```
public class Prog {
    public static void main(String[] args)
    {
        UneFenetre f = new UneFenetre();
        f.ajouterUnBouton("coucou");
    }
}
```

L'héritage

- L'héritage est un des principaux concepts de la programmation par objet
 - En java, on utilise toujours l'héritage lorsque l'on créé une classe
 - Par défaut, la classe hérite par défaut de `Object`
 - Et des méthodes `toString()`, `equals(...)`, etc.
- Pour déclarer explicitement qu'une classe hérite d'une autre, on utilise le mot clé

extends

```
class Truc extends Bidule {  
    ...  
}
```

La classe Truc hérite de la classe Bidule

La classe Truc étend la classe Bidule

La classe Truc est une sous-classe de Bidule

La classe Bidule est la super-classe Truc

Toute instance de Truc sera de type Bidule (et Truc)

Héritage – ajout de membres

- Dans une sous-classe, on peut ajouter
 - Des attributs
 - Des méthodes

Une instance de `Point3D` aura deux attributs `x` et `y` hérité de `Point` mais en plus un attribut `z`

```
class Point {  
    private double x;  
    private double y;  
  
    public double getAbscisse() {  
        return x;  
    }  
  
    public double getOrdonnee() {  
        return y;  
    }  
}
```

```
class Point3D extends Point {  
    private double z;  
  
    public double getProfondeur() {  
        return z;  
    }  
}
```

```
public static void main(String[] args) {
```

```
    Point3D p3d = new Point3D();  
    p3d.getAbscisse();  
    p3d.getOrdonnee();  
    p3d.getProfondeur();  
}
```

Méthodes héritées

Méthode ajoutée

Héritage - redéfinition

- Dans une sous-classe on peut redéfinir des méthodes héritées

```
class Point {  
    private double x;  
    private double y;  
  
    public double getAbscisse() {  
        return x;  
    }  
  
    public double getOrdonnee() {  
        return y;  
    }  
  
    public String toString() {  
        return "("+x+", "+y+")";  
    }  
}
```

On peut aussi le faire avec des attributs mais dans ce cas l'attribut de la super-classe sera caché
C'est déconseillé (et pas très utile)

```
class Point3D extends Point {  
    private double z;  
  
    public double getProfondeur() {  
        return z;  
    }  
  
    public String toString() {  
        return "("+getAbscisse()+", "+getOrdonnee()+  
            ", "+z+")";  
    }  
}
```

La méthode `toString()` héritée de `Point` est redéfinie

```
public static void main(String[] args) {  
    System.out.println(new Point());  
    System.out.println(new Point3D());  
}
```

Qu'affichera le programme ci-dessus ?

Héritage - surcharge

- On peut rajouter des arguments à une méthode héritée : c'est de la surcharge

```
class Point {  
    private double x;  
    private double y;  
  
    public void translate(double dx, double dy) {  
        x+=dx;  
        y+=dy;  
    }  
}
```

La méthode `translate` avec 3 paramètres appelle la méthode `translate` avec 2 paramètres définie dans la super-classe

Le `Point3D` est un `Point` qui a en plus un attribut `z` et une méthode `translate` qui prend 3 paramètres

```
class Point3D extends Point {  
    private double z;  
  
    public void translate(double dx, double dy, double dz) {  
        translate(dx,dy);  
        z+=dz;  
    }  
}
```

Surcharge vs redéfinition

- La surcharge n'est pas de la redéfinition
 - Les deux méthodes coexistent
 - C'est de l'ajout de méthodes

```
Point3D p3d = new Point3D();  
p3d.translate(2.0, 3.0);  
p3d.translate(5.0, 6.0, 8.0);
```

Appel à la méthode `translate` à deux paramètres de `Point`

Appel à la méthode `translate` à trois paramètres de `Point3D`

Héritage et initialisation

- Dans une sous-classe, on se repose sur le constructeur de la super-classe pour initialiser les attributs déclarés dans la super-classe
 - Le constructeur de la super classe est appelé en premier puis celui la classe dérivée

```
public class A {  
    public A() {  
        System.out.println("Dans le constructeur de A");  
    }  
}
```

```
public class TestConstructeursEtHeritage {  
    public static void main(String[] args) {  
        B b = new B();  
    }  
}
```

```
public class B extends A{  
    public B() {  
        System.out.println("Dans le constructeur de B");  
    }  
}
```

Dans le constructeur de A
Dans le constructeur de B

Héritage et initialisation

- Et avec trois classes cela donne quoi ?

```
public class A {  
    public A() {  
        System.out.println("Dans le constructeur de A");  
    }  
}
```

```
public class B extends A {  
    public B() {  
        System.out.println("Dans le constructeur de B");  
    }  
}
```

```
public class C extends B {  
    public C() {  
        System.out.println("Dans le constructeur de C");  
    }  
}
```

```
public class TestConstructeursEtHeritage {  
    public static void main(String[] args)  
    {  
        C truc = new C();  
    }  
}
```

Constructeurs avec paramètres

- Si on veut appeler des constructeurs de la super-classe avec paramètres :
 - il faut les invoquer explicitement via le mot-clé `super`
 - Doit être placé comme première instruction du constructeur

```
public class Personne {
    private String prenom;
    private String nom;

    public Personne(String p , String n) {
        nom=n;
        prenom=p;
    }
    public String toString() {
        return "Je m'appelle "+nom+" "+prenom;
    }
}
```

```
public class Etudiant extends Personne {
    private double motivation;

    public Etudiant(String p, String n, double motiv) {
        super(p, n);
        motivation=motiv;
    }

    public boolean seLeverPourAllerEnCours() {
        return Math.random()<motivation;
    }
}
```

Constructeurs et héritage

- Ecrivez le constructeur avec paramètres de Point3D

```
class Point {  
    private double x;  
    private double y;  
  
    public Point() {  
        this(0,0);  
    }  
    public Point(double abscisse, double ordonnee) {  
        x=abscisse;  
        y=ordonnee;  
    }  
}
```

```
class Point3D extends Point {  
    private double z;  
  
    public Point3D() {  
        this(0,0,0);  
    }  
  
    public Point3D(double abs, double ord, double prof) {  
        Donner la définition  
    }  
}
```

Le mot-clé `super`

- Le mot-clé `super` peut servir à :
 - Appeler un constructeur de la super-classe
 - Appeler explicitement des méthodes de la super-classe

```
class Point {  
    private double x;  
    private double y;  
  
    public String toString() {  
        return "("+x+", "+y+"";  
    }  
}
```

```
class Point3D extends Point {  
    private double z;  
  
    public String toString() {  
        String string2D = super.toString();  
        return string2D.substring(0, string2D.length()-1)+  
            ", "+z+"";  
    }  
}
```

A votre avis qu'est ce qu'il se passe si on omet le mot clé `super` ?

Exception in thread "main" java.lang.StackOverflowError

—► Appel récursif infini à la méthode `toString` de `Point3D...`

La délégation

- La délégation est une relation entre deux classes
 - A mi-chemin entre héritage et composition
 - Pas supporté directement dans Java
 - Mais la plupart des IDE le font automatiquement
- Le principe :
 - On encapsule un objet dans une classe
 - Composition
 - On expose toutes (ou une partie) des méthodes de l'objet encapsulé
 - Cela permet de simuler l'héritage

La délégation - exemple -

- Un cercle est composé d'un centre
 - Un cercle n'est pas un point
 - Mais la translation d'un cercle est déléguée à la translation du centre

```
public class Point {  
    private double x;  
    private double y;  
  
    public Point(double abs, double ord) {  
        x=abs;  
        y=ord;  
    }  
  
    public void translate(double dx, double dy) {  
        x+=dx;  
        y+=dy;  
    }  
}
```

```
public class Cercle {  
    private Point centre;  
    private double rayon;  
  
    public Cercle(Point c, double r) {  
        centre=c;  
        rayon=r;  
    }  
  
    public void translate(double dx, double dy) {  
        centre.translate(dx, dy);  
    }  
}
```

Que se passe t il si on fait une translation d'un point qui est le centre d'un cercle ?

Délégation - exemple -

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

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

    public void translate(double dx, double dy) {
        x+=dx;
        y+=dy;
    }
}
```

```
public class Cercle {
    private Point centre;
    private double rayon;

    public Cercle(Point c, double r) {
        centre=c;
        rayon=r;
    }

    public void translate(double dx, double dy) {
        centre.translate(dx, dy);
    }
}
```

```
public class Delegation {
    public static void main(String[] args) {
        Point p = new Point(1.0,1.0);
        Cercle c = new Cercle(p,2.0);

        p.translate(5, 5);
    }
}
```

p référence le centre du cercle
donc une translation de ce point
va aussi traduire le cercle.

Ce n'est pas forcément désiré.

Quelle solution proposez vous ?

Délégation

- exemple -

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

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

    public Point(Point p) {
        x=p.x;
        y=p.y;
    }

    public void translate(double dx, double dy) {
        x+=dx;
        y+=dy;
    }
}
```

```
public class Cercle {
    private Point centre;
    private double rayon;

    public Cercle(Point c, double r) {
        centre=new Point(c);
        rayon=r;
    }

    public void translate(double dx, double dy) {
        centre.translate(dx, dy);
    }
}
```

```
public class Delegation {
    public static void main(String[] args) {
        Point p = new Point(1.0,1.0);
        Cercle c = new Cercle(p,2.0);

        p.translate(5, 5);
    }
}
```

Solution :

On crée une copie du point qui sert de centre. Comme cela, aucune référence vers le centre n'est visible de l'extérieur.

La translation sur l'objet référencé par p n'affecte pas le cercle

Le modificateur `protected`

- `protected` permet de donner l'accès à un membre (méthode ou attribut) seulement :
 - Aux sous-classes
 - Aux classes du même package
- En règle générale, quand on veut laisser un accès aux sous-classes :
 - utiliser `private` pour les attributs
 - Définir des méthodes d'accès `protected`

Transtypage ascendant

- L'aspect le plus important de l'héritage est la relation de typage
 - Si A extends B alors toute instance de A possède également le type B

```
class Point {  
    private double x;  
    private double y;  
  
    public double getAbscisse() {  
        return x;  
    }  
  
    public double getOrdonnee() {  
        return y;  
    }  
}
```

```
class Point3D extends Point {  
    private double z;  
  
    public double getProfondeur() {  
        return z;  
    }  
}
```

```
public static void main(String[] args) {  
    Point p = new Point3D();  
    // Un Point3D peut être manipulé comme une référence  
    // de type Point  
    p.getAbscisse();  
    p.getOrdonnee();  
    // Par contre dans ce cas, on ne peut appeler que  
    // les méthode déclarés dans la classe Point  
    p.getProfondeur();  
}
```

A quoi sert le transtypage ascendant ?

- On peut écrire du code générique
 - Tout code fonctionnant sur un type donné fonctionnera sur des types plus spécifiques

Composition ou héritage ?

- Doit on utiliser l'héritage ou la composition (délégation) ?
 - L'héritage est à utiliser avec parcimonie !
 - La plupart du temps, on utilise la composition et on délègue certaines méthodes
- Pour décider, il faut se poser la question :
 - Ai-je besoin d'utiliser le transtypage ascendant ?
 - Cela va t-il me permettre de factoriser du code ?