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 7
Un pas de plus vers l'abstraction

- Le mot-clé `final`
- Polymorphisme
- Classes et méthodes abstraites
- Interfaces

Le mot clé final

- Final peut être placé devant
  - Une classe
  - Une méthode
  - Une donnée : un attribut, une variable, un argument
- En fonction de sa place, il a des significations un peu différentes
  - Mais en règle générale, cela veut dire :

    CELA NE PEUT PAS ETRE CHANGE

Rappel – attributs de classe ou attributs statiques

- Les attributs statiques sont partagés par toutes les instances d'une même classe

```
public class CompteEuro {
     private static double tauxChangeDollar=1.24834;
     private static double tauxChangeFrancs=6.55957;

     private Personne titulaire;
     private int soldeEnCentimes;

     public CompteEuro(Personne p, int soldeInitial) {
          titulaire=p;
          soldeEnCentimes=soldeInitial;
     }
     public double soldeEnEuros() {
          return ((double)soldeEnCentimes)/100;
     }
     public double soldeEnDollars() {
          return  ((double)soldeEnCentimes*tauxChangeDollar)/100;
     }
     public double soldeEnFrancs() {
          return  ((double)soldeEnCentimes*tauxChangeFrancs)/100;
     }

     public static void setTauxDeChangeDollar(double newTaux) {
          tauxChangeDollar=newTaux;
     }
}
```

Les taux de change sont partagés par tous les comptes

Le taux de change en Francs ne changera jamais. Ca doit être donc une constante.

Comment déclarer une constante ?

Les données finales

- Pour définir des constantes :
  - i.e. des valeurs (ou références) que l'on ne peut plus changer après la première affectation
  - Exemple : la constante `Math.PI`
    - Elle est déclarée avec les modificateurs **`static`** et **`final`**
    - https://docs.oracle.com/javase/7/docs/api/java/lang/Math.html#PI

```
public class CompteEuro {

     private static double tauxChangeDollar=1.24834;
     private final static double tauxChangeFrancs=6.55957;

     // Etc.
}
```

Les données finales

- Le mot clé `final` n'est pas seulement réservé aux attributs `static`
  - On peut déclarer des attributs d'instance `final`
  - Exemple : Le titulaire d'un compte ne change pas

```
public class CompteEuro {
    private static double tauxChangeDollar=1.24834;
    private static double tauxChangeFrancs=6.55957;

    private final Personne titulaire;
    private int soldeEnCentimes;

    public CompteEuro(Personne p, int soldeInitial) {
        titulaire=p;
        soldeEnCentimes=soldeInitial;
    }
}
```

Attention : cela ne veut pas dire que l'on ne peut pas changer des valeurs à l'intérieur de l'instance de Personne. Cela veut dire que la référence titulaire ne pourra plus être changée (elle référencera toujours la même instance de personne)

Un attribut (d'instance) final ne peut être initialisé qu'au niveau de sa déclaration ou dans les constructeurs.

Méthodes finales

- Une méthode finale ne peut plus être redéfinie dans les sous-classes

```
public class CompteEuro {

    private Personne titulaire;
    private int soldeEnCentimes;

    public CompteEuro(Personne p, int soldeInitial) {
        titulaire=p;
        soldeEnCentimes=soldeInitial;
    }

    public final void deposer(double montantEuros) {
        soldeEnCentimes+=montantEuros*100;
    }
}
```

Je ne veux pas qu'un développeur maladroit redéfinisse cette méthode dans une sous classe pour se prendre une commission au passage

Classes finales

- Une classe finale ne peut pas être étendue
  - On utilise cela pour des questions de préventions d'erreurs ou de sécurité
  - Implicitement toutes ses méthodes sont final

```
public final class MaClasseParfaite {

}
```

- En règle générale, il faut avoir de bonnes raisons de déclarer des méthodes ou classes finales
  - On ne peut plus rien changer après !!!

Le polymorphisme

- Après le notions de classes, d'héritage, le polymorphisme est la plus importante notion
- Le polymorphisme est la capacité d'un code à être utilisé avec différents types

```
class Drone {
     public void lever(int h) {/*...*/}
     public void descendre(int h)
{/*...*/}
     public void tourner(int d){/*...*/}
     public void avancer(int v){/*...*/}
}
```

Est ce que ma télécommande va fonctionner avec un nouveau drone équipé de GPS ?

```
class Telecommande {
     private Drone drone;
     public void associer(Drone d) {
          drone=d;
     }
     public void lever(int h) {
          drone.lever(h);
     }
     public void descendre(int h) {
          drone.descendre(h);
     }
     public void tourner(int d){
          drone.tourner(d);
     }
     public void avancer(int v){
          drone.avancer(v);
     }
}
```

Polymorphisme

- Un code conçu pour fonctionner avec un type donné va fonctionner avec tous ses sous-types

```
class DroneAvecGPS extends Drone {
     private GPS gps;

     public void allerVers(Coordonnee c) {
          /* ... */
     }
}
```

```
public class DesDrones {
     public static void main(String[] args) {
          Telecommande t = new Telecommande();
          DroneAvecGPS d = new DroneAvecGPS();
          t.associer(d);
          t.lever(10);
     }
}
```

```
class Telecommande {
     private Drone drone;
     public void associer(Drone d) {
          drone=d;
     }
     public void lever(int h) {
          drone.lever(h);
     }
     public void descendre(int h) {
          drone.descendre(h);
     }
     public void tourner(int d){
          drone.tourner(d);
     }
     public void avancer(int v){
          drone.avancer(v);
     }
}
```

La télécommande va fonctionner avec le DroneAvecGPS !!!

Type de la référence et type réel

- Une référence a un type et un objet a également un type
  - Avec l'héritage, les types peuvent être différents

```
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();
}
```

p est une référence de type Point, mais l'objet référencé est de type Point3D (type réél)

Liaison dynamique

- Le choix de code de méthode appelé est fait lors de l'exécution
  - Java détermine le type des objets référencés lors de l'exécution afin de savoir quel code de méthode sera appelé
    - Sauf pour les méthodes final (private) et static

Transtypage descendant

- On peut avoir parfois besoin de tester le type précis d'un objet référencé
  - Exemple : la méthode equals(Object o)

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

     public double getAbscisse() {
          return x;
     }
     public double getOrdonnee() {
          return y;
     }
     public boolean equals(Object o) {
          if (o instanceof Point) {
               Point p = (Point) o;
               return p.x==x && p.y==y;
          }
          return false;
     }
}
```

Le transtypage descendant n'est pas sûr. Il faut vérifier avant d'effectuer le transtypage

Les classes abstraites

- Rappels : l'héritage est fait pour
  - Factoriser du code
    - On met le code commun à toutes le sous-classes dans une super classe
  - Factoriser des interfaces
    - On défini dans la super-classe toutes les signatures de méthodes communes sur les sous-classes
- Il arrive que l'on ne puisse pas définir le corps d'une méthode…

Avant héritage...

```java
class Rectangle {
     private int largeur;
     private int hauteur;

     public double aire() {
          return largeur*hauteur;
     }
}
```

Je veux ajouter une classe Cercle…

```java
class Cercle {
     private int rayon;

     public double aire() {
          return Math.PI*rayon*rayon;
     }
}
```

```java
class Dessin {
     private Rectangle[] rect;
     private Cercle[] cercles;

     public double aireTotale() {
          double total=0;
          for (Rectangle r : rect) {
               total+=r.aire();
          }
          for (Cercle c : cercles) {
               total+=c.aire();
          }
          return total;
     }
}
```

C'est MOCHE !!! Pas maintenable, pas générique
Et si on introduisait une super-classe commune Forme ?

Avec l'héritage

```
class Forme {
     public double aire() {
          //
     }
}
```

```
class Rectangle extends Forme {
     private int largeur;
     private int hauteur;

     public double aire() {
          return largeur*hauteur;
     }
}
```

```
class Cercle extends Forme {
     private int rayon;

     public double aire() {
          return Math.PI*rayon*rayon;
     }
}
```

```
class Dessin {
     private Forme[] formes;

     public double aireTotale() {
          double total=0;
          for (Forme f : formes) {
               total+=f.aire();
          }
          return total;
     }
}
```

Par contre qu'est ce que je mets ici ?

La classe Forme est trop générale pour pouvoir définir la méthode aire()

Le mot-clé abstract

- La solution est
  - De ne pas définir le contenu de la méthode aire()
  - Et de dire que la classe ne peut pas être instanciée
  - C'est possible grâce au mot-clé **`abstract`**

```
abstract class Forme {
    public abstract double aire();
}
```

Le modificateur abstract

- Il se place devant :
  - Des méthodes que l'on ne peut pas définir au niveau de la classe
  - Et devant une classe qui ne peut pas être instanciée
- Une classe qui possède au moins une méthode abstraite doit être déclaré abstraite

Les interfaces

- La notion d'interface en Java permet de pousser plus loin le concept d'abstraction
- Une classe abstraite peut contenir des méthodes implémentés
- Une interface peut être vue comme une classe complètement abstraite
  - On défini seulement les signatures de méthodes
    - Nom de méthode, type de retour, liste d'arguments
    - Mais jamais le corps de la méthode
  - Et éventuellement des constantes

Pourquoi des interfaces et pas d'héritage multiple ?

```
class Etudiant {
    void faireLaFete() {/*beaucoup*/}
    void travailler() {/*mais pas trop*/}
    void allerEnCours() {/*parfois*/}
}
```

```
class Salarie {
    void travailler() {/*beaucoup*/}
    void allerAuTravail() {/*tous les
jours*/}
}
```

Si je veux définir une classe `EtudiantSalarie`, quelle est la méthode héritée ? `travailler()` de `Etudiant` ou celle de `Salarie`

A cause de ce genre d'ambiguité, Java n'autorise pas l'héritage multiple

Avec les interfaces ce problème ne se pose pas, on n'hérite pas du code mais juste de la signature

Motivation

```
abstract class Animal {
      abstract void crier();
}
```

```
abstract class Felin extends Animal {
   void griffer() {
      System.out.println("Aie!");
   }
}
```

```
abstract class Canin extends Animal {
   void mordre() {
            System.out.println("Aie!");
   }
}
```

```
class Chat extends Felin {
   void crier() {
      System.out.println("Miaou");
   }
}
```

```
class Chien  extends Canin {
   void crier() {
      System.out.println("Whouaf");
   }
}
```

```
class Tigre extends Felin {
   void crier() {

System.out.println("Ahrrfff !");
   }
}
```

```
class Loup  extends Canin {
   void crier() {
      System.out.println("Ouuuh");
   }
}
```

Comment définir la notion d'animal de compagnie ?
Méthodes : faireDesCalins() et jouer()

Option 1 : Ajouter les méthodes à Animal

```
abstract class Animal {
   abstract void crier();
   abstract void faireDesCalins();
   abstract void jouer();
}
```

Quel est le problème ???

Pensez vous que l'on, peut faire des câlins à un loup ou à un tigre ?

Option 2 : Ajouter les méthodes uniquement à Chien et Chat

```
class Chat extends Felin {
   void crier() {
      System.out.println("Miaou");
   }
   void faireDesCalins() {
      // ...
   }
   void jouer() {
      // ...
   }
}
```

```
class Chien  extends Canin {
   void crier() {
      System.out.println("Whouaf");
   }
   void faireDesCalins() {
      // ...
   }
   void jouer() {
      // ...
   }
}
```

```
class Personne {
     Chien[] mesChiens;
     Chat[] mesChats;
     // ... Contructeurs et autre ...
     void quandOnSEnnuie() {
          for (Chat c : mesChats) {
               c.jouer();
          }
          for (Chien c : mesChiens) {
               c.jouer();
          }
     }
}
```

Admettons maintenant que l'on ajoute une classe CochonDeCompagnie …
cela devient fastidieux

Option 3 : l'interface Compagnon

```
interface Compagnon {
     void jouer();
     void faireDesCalins();
}
```

```
class Chat extends Felin implements Compagnon {
   void crier() {
      System.out.println("Miaou");
   }
   void faireDesCalins() {
      // ...
   }
   void jouer() {
      // ...
   }
}
```

```
class Chien  extends Canin  implements Compagnon {
   void crier() {
      System.out.println("Whouaf");
   }
   void faireDesCalins() {
      // ...
   }
   void jouer() {
      // ...
   }
}
```

Les chiens sont à la fois Animal et Compagnon

Une interface peut être vue comme un type. Dans ce cas les deux seules méthodes disponibles sont :

- jouer()
- faireDesCalins() ;

```
class Personne {
     Compagnon[] mesCompagnons;
     // ... Contructeurs et autre ...
     void quandOnSEnnuie() {
          for (Compagnon c : mesCompagnons)
{
               c.jouer();
          }
     }
}
```

Généralités sur les interfaces

- Une interface peut SEULEMENT contenir :
  - Des signatures de méthodes
    - i.e. pas de corps de méthodes
    - pas de constructeurs
  - Des constantes (attributs déclarés static final)
- Une interface peut étendre une autre interface
  - Mais pas de classe

```
interface A {
    void m1();
}

interface B extends A {
    void m2();
}
```

```
class M implements B {

    public void m1() {
        // Corps de la méthode
    }

    public void m2() {
        // Corps de la méthode
    }
}
```

Généralités sur les interfaces

- Une interface peut hériter d'une ou plusieurs interfaces
  - Pas de pb d'héritage de corps de méthode

```
interface A {
     void m1();
}
```

```
interface B  {
     void m2();
}
```

```
interface C extends A,B {
     void m3();
}
```

- Une classe peut implémenter plusieurs interfaces

```
class M implements A, B {

     public void m2() {
          // corps de méthode
     }
     public void m1() {
          // corps de méthode
     }
}
```

Quelques interfaces de la bibliothèque Java

- Il existe deux utilisations distinctes des interfaces en Java
  - Les interfaces classiques qui définissent des signatures de méthodes
    - `Comparable` → `compareTo(…)`
    - `java.util.Iterator` → `hasNext(), next(), remove(...)`
  - Les interfaces de « balisage »
    - Elles ne définissent aucune signature de méthodes
    - Elles permettent juste d'indiquer que les classes implémentant ces interfaces ont certaines propriétés
      - `Cloneable, Serializable`

L'interface Comparable

- Cette interface avait la définition suivante dans les versions de java < 1.5

```
public interface Comparable {
    public int compareTo(Object o);
}
```

  - y.compareTo(x) retourne un entier
    - <0 , si y<x
    - =0, si x=y
    - >0 si y>x
- Elle est implémentée par de nombreuses classes de la librairie standard Java
  - String, Integer (et les autres), etc.

Exercice : Créer une classe Personne qui compare deux personnes selon l'ordre alphabétique de leur nom puis de leur prénom.

https://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html

L'interface Cloneable

- Sa définition est la suivante :

```
public interface Cloneable {
}
```

  - Elle indique seulement que l'appel à la méthode clone() (héritée de la classe Object) est légal
    - i.e. que ca ne va pas faire d'erreur à l'exécution
      - CloneNotSupportedException

https://docs.oracle.com/javase/7/docs/api/java/lang/Cloneable.html