Programmation Orientée Objet : Classes abstraites et interfaces

E. Hyon, V. Bouquet[1]

[1]valentin.bouquet@parisnanterre.fr

Licence MIASHS Miage - 2023/2024

Grandes lignes du cours

Classes abstraites

- Problème à résoudre
- Définition

Interfaces

- Préambule et définition
- Déclaration et implémentation
- Cas d'utilisation

Interface ou classe abstraite ?

Outline

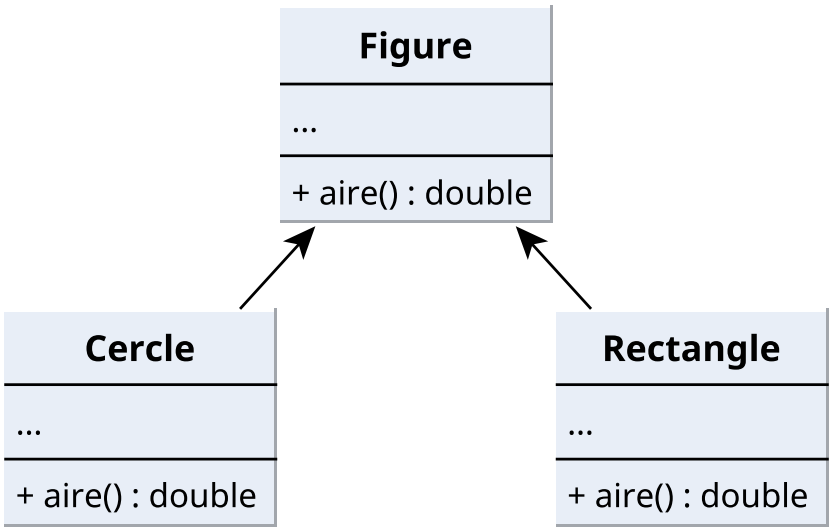
Classes abstraites
- Problème à résoudre
- Définition

Interfaces
- Préambule et définition
- Déclaration et implémentation
- Cas d'utilisation

Interface ou classe abstraite ?

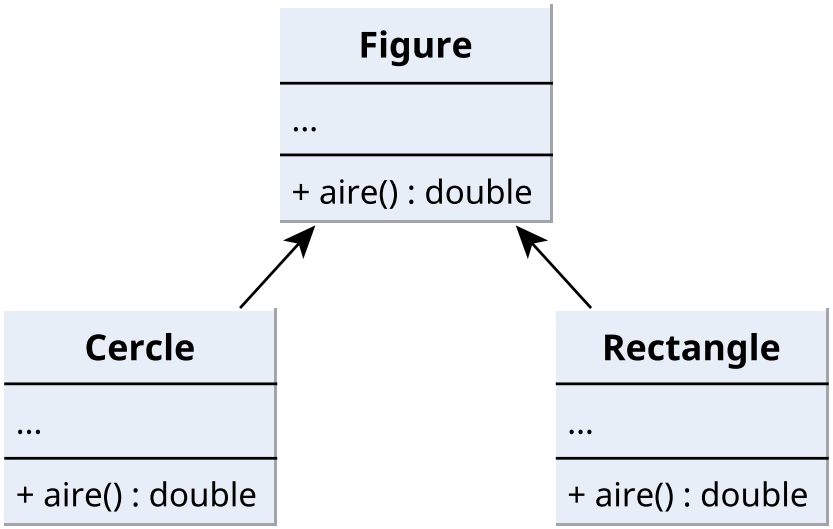
Exemple : besoin

Considérons la hiérarchie de classes suivantes :



Cahier des charge : toutes les classes disposent de la méthode `aire()` retournant l'aire de la figure géométrique définie par la classe.

Exemple : problème



La méthode `aire()` ne peut pas être implémentée dans la déclaration de la classe `Figure`.

En effet, on ne sait pas comment calculer l'aire d'une figure quelconque et l'aire d'un rectangle ne se calcule pas de la même manière que l'aire d'un cercle.

Exemple : solution

Une manière de procéder est de donner la signature de la méthode `aire()` mais **pas son implémentation** dans la classe `Figure`. Une telle méthode est appelée une méthode abstraite.

Pour déclarer une méthode abstraite, on utilise le mot-clef **abstract** :

```
abstract public double aire();
```

C'est à la charge des concepteurs des classes `Cercle` et `Rectangle` d'implémenter cette méthode permettant ainsi de spécifier un comportement différent pour chaque classe.

Outline

Classes abstraites
- Problème à résoudre
- Définition

Interfaces
- Préambule et définition
- Déclaration et implémentation
- Cas d'utilisation

Interface ou classe abstraite ?

Définition

Un classe contenant au moins une méthode abstraite est appelée une classe abstraite.

Les classes abstraites doivent être déclarées à l'aide du mot-clef **abstract**

```
abstract public class Figure {
    private String nom;
    public Figure(String nom) {this.nom=nom;}
    abstract public double aire();
    public String quiSuisJe()
    {System.out.println("Je␣suis␣un␣" + this.nom);}
}
```

Exemples : implémentation de la méthode abstraite

```
public class Cercle extends Figure {
  private double rayon;
  public Cercle(double rayon)
  {super("cercle"); this.rayon = rayon;}
  public double aire()
  {return Math.PI * this.rayon * this.rayon;}}

public class Rectangle extends Figure {
  private double largeur;
  private double longueur;
  public Rectangle(double largeur,double longueur)
  {super("rectangle");this.largeur = largeur;
    this.longueur = longueur;}
  public double aire()
  {return this.largeur * this.longueur;}}
```

À retenir

- Si au moins une des méthodes d'une classe est abstraite alors c'est une classe abstraite.
- Une classe fille héritant d'une classe mère abstraite doit implémenter toutes ses méthodes abstraites, sinon elle est aussi une classe abstraite !
- On ne peut pas instancier une classe abstraite car au moins une des ses méthodes n'a pas d'implémentation.
- Même si la classe n'est jamais instanciée il est préferable de définir un constructeur pour initialiser les attributs privés (appel à super).

```java
public abstract class Polygone extends Figure {

    private int nombreCote;

    public Polygone(String nom, int nombreCote) {
      super(nom);
      this.nombreCote = nombreCote;
    }

    public abstract double aire();

    public String quiSuisJe(){
      System.out.println("Je suis un polygone"
                                + getNom());
    }
}
```

À retenir (suite)

- Une méthode abstraite publique fait partie de la partie publique de la classe abstraite.
- Elle peut donc être invoquée même si ce sera la méthode de l'objet réellement instancié qui sera utilisée.
- Une méthode abstraite publique peut être invoquée dans d'autres méthodes au sein de la classe abstraite.
- Le principal intérêt de la classe abstraite est de regrouper le code commun à toutes les classes d'une même descendance dans une même classe.

À retenir (suite)

Exemple

```
abstract public class Figure {

    private String nom;
    public Figure(String nom) {this.nom = nom;}

    abstract public double aire();
    abstract public double perimetre();

    public String quiSuisJe()
    {System.out.println("Je␣suis␣un␣" + this.nom);}

    public double ratio(){
        return aire()/perimetre();}
}
```

Outline

Classes abstraites
- Problème à résoudre
- Définition

Interfaces
- Préambule et définition
- Déclaration et implémentation
- Cas d'utilisation

Interface ou classe abstraite ?

Problème à résoudre :

- Assurer qu'un ensemble de classes offre un service minimum commun.
- Faire du polymorphisme avec des objets dont les classes n'appartiennent pas à la même hiérarchie d'héritage.
- Utilisation d'objets sans connaître leurs types réels.

Solution : la définition d'un type complètement abstrait nommé **interface** (notion de **contrat**).

Définition

Quand **toutes les méthodes** d'une classe sont **abstraites** et qu'il n'y a **aucun attribut**, on aboutit à la notion d'interface.

Définition

- *Une interface est un prototype de classe. Elle définit uniquement la signature de méthodes (et éventuellement de constantes).*
- *Le corps des méthodes dont l'en tête est donné dans une interface doivent être **implémentées** dans les classes construites à partir de ce prototype.*
- *Une interface est une "classe" purement abstraite dont toutes les méthodes sont abstraites et publiques et sans attributs (les mots-clefs **abstract** et **public** sont optionnels (implicites)).*

Outline

Classes abstraites
- Problème à résoudre
- Définition

Interfaces
- Préambule et définition
- Déclaration et implémentation
- Cas d'utilisation

Interface ou classe abstraite ?

Déclaration : syntaxe

La définition d'une interface se présente comme celle "d'une classe purement abstraite" en utilisant le mot-clef `interface` à la place du mot-clef `class`

```
interface MonInterface {

  void uneMethode();

  void uneSecondeMethode();

}
```

Cela définit un type du nom de l'interface.

Exemples

```
interface Comparable {
  boolean compareTo(Object o);
}

interface Figure{
  void dessineToi();
  void deplaceToi(int x,int y);
}
```

NB : **public** et **abstract** sont implicites devant les déclarations de méthodes.

Implémentation d'une interface

Le contrat proposé dans une interface doit être mis en oeuvre (et codé) au sein de classes spécifiques et concrètes.

Pour préciser qu'une classe implémente une interface, on utilise le mot-clef `implements` :

```
class Classe implements Interface { ... }
```

Attention : la classe doit implémenter **toutes** les méthodes de l'interface, sinon elle doit être déclarée **abstract**.

Attention : une interface peut **étendre** une ou plusieurs interfaces. Il faut utiliser le mot clef extends.

Implémentation d'une interface : Exemple 1

```
public class A implements Comparable {
  private int x;
  private int y;
  public A(int x, int y){
    this.x = x;
    this.y = y;
  }

  boolean compareTo(Object o){
    if (o instance of A)
      A a = (A) o; // downcast
      if (this.x + this.y >= a.getX() + a.getY())
          return true;
    return false;
  }
}
```

Implémentation d'une interface : Exemple 2

```
public class Ville implements Comparable {
  private int population;
  private String nom;
  public Ville(String nom, int population){
    this.nom = nom;
    this.population = population;
  }

  boolean plusGrand(Object o){
    if (o instance of Ville)
      Ville v = (V) o; \\ downcast
      if (population >= a.getPopulation() )
          return true;
    return false;
  }
}
```

Implémentation d'une interface (suite)

Une classe peut implémenter plusieurs interfaces :

```
class A implements Interface1, Interface2, ...
```

Une classe peut hériter d'une autre classe et implémenter une ou plusieurs interfaces :

```
class B extends A implements Interface1, Interface2, ...
```

Outline

Classes abstraites
- Problème à résoudre
- Définition

Interfaces
- Préambule et définition
- Déclaration et implémentation
- Cas d'utilisation

Interface ou classe abstraite ?

Utilisation comme types de données

- Une interface est considérée comme un type et peut remplacer une classe pour la déclaration d'une variable, un paramètre, une valeur de retour, etc.
- Par exemple,

```
Comparable v1,v2;

v1 = new Ville("Nanterre",93742)
v2 = new Autre(0);
```

Cela indique que les variables `v1, v2` contiendront des références vers des objets d'une classe implémentant l'interface `Comparable`

Cast et interfaces

On peut toujours faire des casts (upcast et downcast) entre une classe et une interface qu'elle implémente (et un upcast entre une interface et la classe Object) :

```
// Ville implements Comparable
// upcast (implicite) Ville -> Comparable
Comparable c1 = new Ville("Cannes", 74285);
Comparable c2 = new Ville("Nice", 342522);

//upcast (implicite) Comparable -> Object
if (c1.compareTo(c2))

// downcast (explicite) Comparable ->  Ville
Ville v = (Ville) c2;
System.out.println(c2.nbHabitants());
```

Cast et interfaces

Une classe B qui hérite de A avec A qui implémente l'interface I peut aussi etre déclarée comme étant de type I

```
public class A implements I {
   ...
}
public class B extends A {....}

public void main(String[] args) {
   I ref= new B();

   ref instanceof I; // true
   ref instanceof A; // true
}
```

À retenir

- Les interfaces ne sont pas instanciables.
- Une interface n'a pas de constructeurs.
- Les attributs d'une interface sont toujours publiques, statiques et final (ceci est implicite). Les attributs sont donc toujours des constantes

```
public static final float PI = 3.14f;
```

- Il est préférable de ne donner que la signature des méthodes (même si il est possible de déclarer des méthodes privées et `default`). Le abstract et le public sont implicites :

```
abstract public int f();
```

- Une interface peut hériter d'une autre interface :

```
interface Interface1 extends Interface2
```

Classe abstraite ou interface ?

Il est préférable d'utiliser une interface quand cela est possible !
Pourquoi :

- Il est facile de faire implémenter une interface a deux classes existantes (aucune limite sur le nombre d'implémentation).
- Les interfaces encouragent la programmation par contrat, ce qui conduit à une meilleure encapsulation des classes et à une séparation claire des préoccupations.
- Les interfaces sont très utiles pour ajouter des comportements additionnels (mixins). Par exemple pour les logs, la comparaison d'objets, le formatage de texte, gestion des états, gestion de fichiers.

Classe abstraite ou interface ? (2)

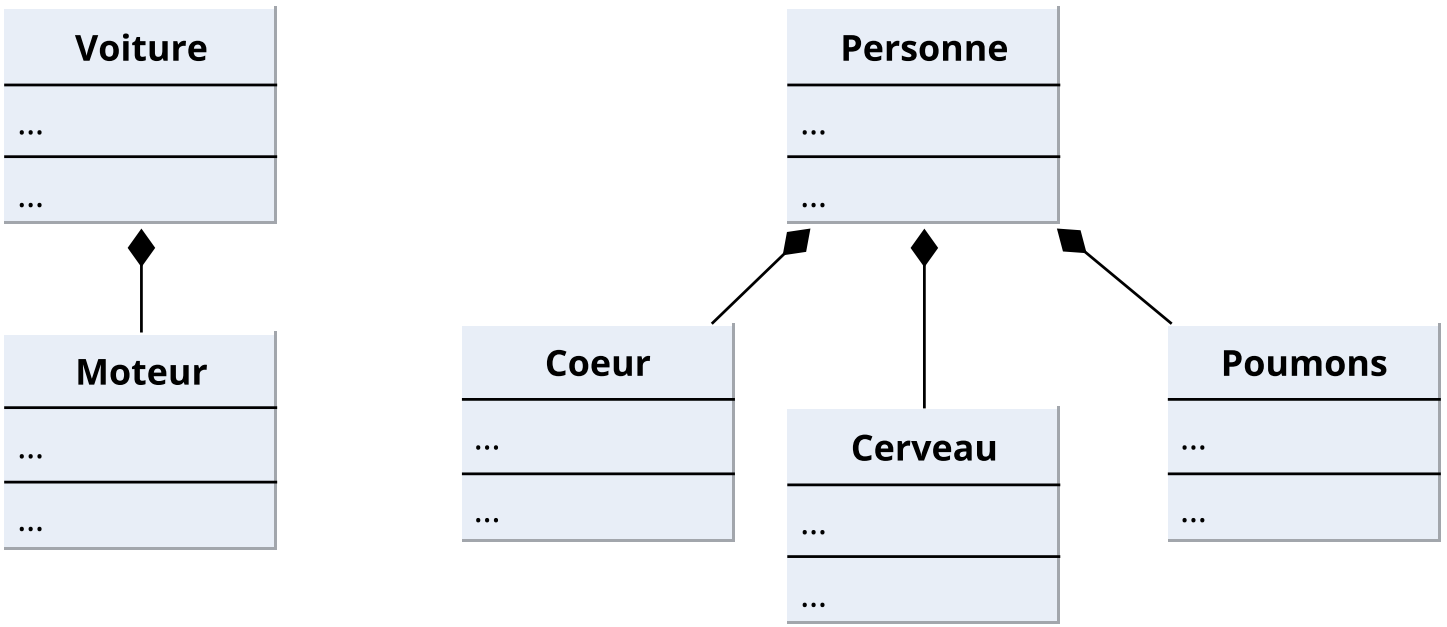
Question : mais je souhaite réutiliser le code de la même méthode dans plusieurs classes. Comment faire ?

- Si cette méthode ne dépend pas de l'état de l'objet : il faut utiliser une méthode statique !
- Il est toujours possible d'utiliser une classe abstraite...
- Mais il existe aussi la composition ! Il est d'ailleurs préférable d'utiliser la composition que d'étendre une classe (quand cela est possible).

Concept de composition

Principe : un objet peut incorporer d'autres objets
C'est tout...

Des exemples :



Exemple de composition : la classe Moteur

Définition d'une classe Moteur :

```
class Moteur {
  private String type;

  public Moteur(String type) {
      this.type = type;
  }

  public void demarrer() {
      System.out.println("Demarrage du moteur " + type);
  }
}
```

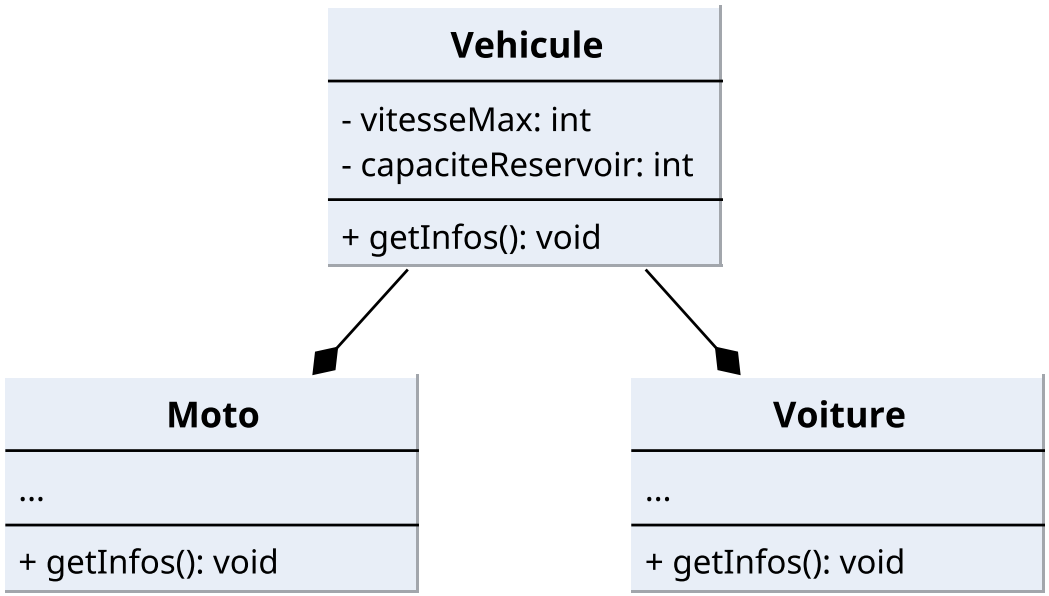
Exemple de composition : La classe Voiture

```
class Voiture {
  private String modele;
  private Moteur moteur;

  public Voiture(String modele, Moteur moteur) {
      this.modele = modele;
      this.moteur = moteur;
  }

  public void rouler() {
      System.out.println("La voiture de modele "
                         + modele + " demarre.");
      moteur.demarrer();
      System.out.println("La voiture roule.");
  }
}
```

Exemple de composition : Véhicules



Exemple de composition : la classe Vehicule

```
class Vehicule {
  private int vitesseMax;
  private int capaciteReservoir;

  public Vehicule(int vM, int cR) {
      this.vitesseMax = vM;
      this.capaciteReservoir = cR;
  }

  public void getInfos() {
      System.out.println("Vitesse maximale : "
              + vitesseMax + " km/h");
      System.out.println("Capacite du reservoir : "
              + capaciteReservoir + " litres");
  }
}
```

Exemple de composition : la classe Voiture

```
class Voiture {
  private Vehicule vehicule;

  public Voiture(int vM, int cR) {
      vehicule = new Vehicule(vM, cR);
  }

  public void getInfos() {
    vehicule.getInfos();
    // on peut aussi afficher d'autres informations
    // propre a la voiture
  }
}
```

Exemple de composition : la classe Moto

```
class Moto {
  private Vehicule vehicule;

  public Moto(int vM, int cR) {
      vehicule = new Vehicule(vM, cR);
  }

  public void getInfos() {
    vehicule.getInfos();
    // on peut aussi afficher d'autres informations
    // propre a la moto
  }
}
```

Pourquoi utiliser la composition ?

- Réutilisation du code : on peut combiner plusieurs objets pour former un objet plus complexe.
- Flexibilité : avec la composition, on peut plus facilement changer les méthodes d'une classe. On évite aussi la rigidité de l'héritage.
- Encapsulation : on limite l'accès aux détails des composants internes. Chaque objet est responsable de ses données. On est pas obligé d'exposer toutes les méthodes des objets.
- Facilité des tests : on peut tester chaque objet de manière isolé (notions de test logiciel en master).