

Héritage (2)

Programmation Orientée Objet

Jean-Christophe Routier
Licence mention Informatique
Université Lille 1



Université
de Lille
1 SCIENCES
ET TECHNOLOGIES



UFR IEEA
Formations en
Informatique de
Lille 1



Exemple : Dessin de Formes

- on souhaite réaliser une application de dessin permettant, entre autres, la manipulation de différentes formes graphiques : cercles, triangles, rectangles, polygônes, carrés, etc.
- un objet Canvas gère l'état courant du dessin,
- l'application dispose d'outils permettant de déplacer la forme sélectionnée, d'en changer la taille, de lui appliquer une rotation, etc.

Canvas et chacune des classes décrivant les outils doivent pouvoir être implémentées **indépendamment** des formes existantes

↪ on doit pouvoir **facilement** ajouter des formes à l'application.

(Open Close Principle)

```
public class Canvas {
    protected List<Shape> lesFormes;
    public void repaint() {
        for(Shape s: this.lesFormes) {
            s.draw();
        }
    }
}

public class MoveTool {
    public void move(Shape selectedShape) {
        selectedShape.move(...);
    }
}

public class RotateTool {
    public void rotate(Shape selectedShape) {
        selectedShape.rotate(...);
    }
}

public class ResizeTool {
    public void resize(Shape selectedShape) {
        selectedShape.resize(...);
    }
}
```

Il est nécessaire de disposer d'un type **Shape**

Interface ou héritage ?

interface il n'est pas possible de définir un comportement pour les méthodes `draw`, `rotate`, `resize` ou d'autres de `Shape`.

héritage un certain nombre de comportements et d'attributs peuvent être définis de manière commune pour toutes les formes : couleur du trait, motif du remplissage, épaisseur du trait, gestion du plan de profondeur d'affichage, etc.

Interface ou héritage ?

interface il n'est pas possible de définir un comportement pour les méthodes `draw`, `rotate`, `resize` ou d'autres de `Shape`.

héritage un certain nombre de comportements et d'attributs peuvent être définis de manière commune pour toutes les formes : couleur du trait, motif du remplissage, épaisseur du trait, gestion du plan de profondeur d'affichage, etc.

Autre possibilité, utiliser une approche hybride :

une **classe abstraite**

Classes abstraites

Une classe abstraite

est une classe (comporte donc des attributs et des méthodes),

Classes abstraites

Une classe abstraite

est une classe (comporte donc des attributs et des méthodes),
déclare de méthodes sans comportement attaché, ces méthodes sont
dites **méthodes abstraites**,

Classes abstraites

Une classe abstraite

est une classe (comporte donc des attributs et des méthodes),
déclare de méthodes sans comportement attaché, ces méthodes sont
dites **méthodes abstraites**,
ne peut pas créer d'instance (même si elle peut déclarer un
constructeur),

Classes abstraites

Une classe abstraite

est une classe (comporte donc des attributs et des méthodes),
déclare de méthodes sans comportement attaché, ces méthodes sont
dites **méthodes abstraites**,

ne peut pas créer d'instance (même si elle peut déclarer un
constructeur),

les classes qui en héritent doivent concrétiser le comportement de
toutes les méthodes abstraites

↪ sous peine d'être elles aussi abstraites

abstract

- déclaration classe abstraite : ajouter le qualificatif **abstract** avant `class` dans l'entête de déclaration.

```
public abstract class SomeAbstractClass ... {  
    ...  
}
```

abstract

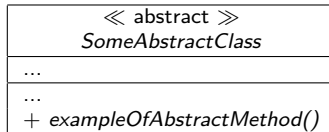
- déclaration classe abstraite : ajouter le qualificatif **abstract** avant `class` dans l'entête de déclaration.
- déclaration méthode abstraite : ajouter le qualificatif **abstract** avant le type de retour dans la signature de la méthode et ne donner aucun corps.

```
public abstract class SomeAbstractClass ... {  
    ...  
    public abstract Type exampleOfAbstractMethod(...);  
}
```

abstract

- déclaration classe abstraite : ajouter le qualificatif **abstract** avant `class` dans l'entête de déclaration.
- déclaration méthode abstraite : ajouter le qualificatif **abstract** avant le type de retour dans la signature de la méthode et ne donner aucun corps.

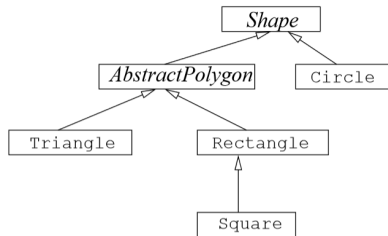
```
public abstract class SomeAbstractClass ... {  
    ...  
    public abstract Type exampleOfAbstractMethod(...);  
}
```



Retour en Forme

```
public abstract class Shape {
    protected int lineWidth;
    protected FillStyle fillStyle;
    protected byte depth;
    public void setLineWidth(int width) { this.lineWidth = width }
    public void toFront() { depth = 0; }
    public FillStyle getFillStyle() { return fillStyle; }
    public abstract void draw();
    public abstract void rotate(double angle);
    public abstract void move(...);
    public abstract void resize(...);
}

public class Circle extends Shape {
    protected Point center;
    protected int radius;
    public void draw() { ... code ... }
    public void rotate(double angle) { ... code ... }
    public void move(...) { ... code ... }
    public void resize(...) { ... code ... }
    public Point getCenter() { ... code ... }
}
```



```

public abstract class Shape { ... }

public class Circle extends Shape { ... }
public abstract class AbstractPolygon extends Shape {
    protected List<Edge> edges;
    public abstract int getNumberOfEdges();
    ...
}
// les deux classes suivantes fournissent une définition pour getNumberOfEdges()
// (et les autres méthodes abstraites de Shape)
public class Triangle extends AbstractPolygon { ... }
public class Rectangle extends AbstractPolygon { ... }
// réutilise le getNumberOfEdges() de Rectangle
public class Square extends Rectangle { ... }

```

personnages de jeu

```
public abstract class GameCharacter {
    protected static final int DEFAULT_LIFEPOINTS = 12;
    protected static final int DEFAULT_STRENGTH = 8;
    protected int lifePoints;
    protected int strength;
    public GameCharacter() { this(GameCharacter.DEFAULT_LIFEPOINTS, GameCharacter.DEFAULT_STRENGTH); }
    public GameCharacter(int lifePoints, int strength) {
        this.lifePoints = lifePoints; this.strength = strength;
    }
    public abstract void interactWith(GameCharacter other);
    public int isDead() { return this.lifePoints <= 0; }
    public int removeLifePoints(int lp) { return this.lifePoints -= lp; }
    public int strength() { return this.strength; }
    public int flee() { ... }
    public void attack(GameCharacter other) { other.removeLifePoints(this.strength); }
}
```

personnages de jeu

```

public abstract class GameCharacter {
    protected static final int DEFAULT_LIFEPOINTS = 12;
    protected static final int DEFAULT_STRENGTH = 8;
    protected int lifePoints;
    protected int strength;
    public GameCharacter() { this(GameCharacter.DEFAULT_LIFEPOINTS, GameCharacter.DEFAULT_STRENGTH); }
    public GameCharacter(int lifePoints, int strength) {
        this.lifePoints = lifePoints; this.strength = strength;
    }
    public abstract void interactWith(GameCharacter other);
    public int isDead() { return this.lifePoints <= 0; }
    public int removeLifePoints(int lp) { return this.lifePoints -= lp; }
    public int strength() { return this.strength; }
    public int flee() { ... }
    public void attack(GameCharacter other) { other.removeLifePoints(this.strength); }
}

public class Orc extends GameCharacter {
    protected static final int ORC_LIFEPOINTS = 10;
    protected PersonneId pid;
    public Orc(int strength) { super(ORC_LIFEPOINTS, strength); }
    public void interactWith(GameCharacter other) { this.attack(other); } // moi vois, moi tue
    ...
}

```


personnages de jeu

```

public abstract class GameCharacter {
    protected static final int DEFAULT_LIFEPOINTS = 12;
    protected static final int DEFAULT_STRENGTH = 8;
    protected int lifePoints;
    protected int strength;
    public GameCharacter() { this(GameCharacter.DEFAULT_LIFEPOINTS, GameCharacter.DEFAULT_STRENGTH); }
    public GameCharacter(int lifePoints, int strength) {
        this.lifePoints = lifePoints; this.strength = strength;
    }
    public abstract void interactWith(GameCharacter other);
    public int isDead() { return this.lifePoints <= 0; }
    public int removeLifePoints(int lp) { return this.lifePoints -= lp; }
    public int strength() { return this.strength; }
    public int flee() { ... }
    public void attack(GameCharacter other) { other.removeLifePoints(this.strength); }
}

public class Orc extends GameCharacter {
    protected static final int ORC_LIFEPOINTS = 10;
    protected PersonneId pid;
    public Orc(int strength) { super(ORC_LIFEPOINTS, strength); }
    public void interactWith(GameCharacter other) { this.attack(other); } // moi vois, moi tue
    ...
}

public class CarefulCharacter extends GameCharacter {
    protected int cautionThreshold;
    public CarefulCharacter(int strength, limit) { super(); this.cautionThreshold = limit; }
    public void interactWith(GameCharacter other) { // inutile de prendre des risques
        if (other.strength() - this.strength > this.cautionThreshold) { this.flee(); }
        else { this.attack(other); }
    } ...
}

```

Héritage d'interfaces

- il est possible de définir une interface héritant d'une autre interface.
- comme pour les classes, on peut *étendre* par héritage.

```
public interface Aquatic {  
    public void swim();  
}  
public interface Marine extends Aquatic {  
}  
public interface Predator {  
    public void hunt();  
}  
public interface MarinePredator extends Marine, Predator {  
    public float kiloOfFishEatenADay();  
}
```

- intérêt : typage mutiple = intersection de types.

```
public interface Animal { }  
public interface Aquatic { }
```

```
public interface Animal { }  
public interface Aquatic { }  
public interface AquaticAnimal extends Animal, Aquatic { }
```

```
public interface Animal { }  
public interface Aquatic { }  
public interface AquaticAnimal extends Animal, Aquatic { }  
public class Zoo {  
    public void addAnimal(Animal animal) { ... }  
}
```

```
public interface Animal { }  
public interface Aquatic { }  
public interface AquaticAnimal extends Animal, Aquatic { }  
public class Zoo {  
    public void addAnimal(Animal animal) { ... }  
}  
public class Aquarium {  
    public void add(Aquatic aquatic) { ... }  
}
```

```

public interface Animal { }
public interface Aquatic { }
public interface AquaticAnimal extends Animal, Aquatic { }
public class Zoo {
    public void addAnimal(Animal animal) { ... }
}
public class Aquarium {
    public void add(Aquatic aquatic) { ... }
}
public class ZooWithAquarium extends Zoo {
    protected Aquarium aquarium;
    public void addAquaticAnimal(AquaticAnimal aquAnimal) {
        this.addAnimal(aquAnimal);
        this.aquarium.add(aquAnimal);
    }
    ...
}

```

```

public interface Animal { }
public interface Aquatic { }
public interface AquaticAnimal extends Animal, Aquatic { }
public class Zoo {
    public void addAnimal(Animal animal) { ... }
}
public class Aquarium {
    public void add(Aquatic aquatic) { ... }
}
public class ZooWithAquarium extends Zoo {
    protected Aquarium aquarium;
    public void addAquaticAnimal(AquaticAnimal aquAnimal) {
        this.addAnimal(aquAnimal);
        this.aquarium.add(aquAnimal);
    }
    ...
}
public class Mammal implements Animal { ...}

```



```

public interface Animal { }
public interface Aquatic { }
public interface AquaticAnimal extends Animal, Aquatic { }
public class Zoo {
    public void addAnimal(Animal animal) { ... }
}
public class Aquarium {
    public void add(Aquatic aquatic) { ... }
}
public class ZooWithAquarium extends Zoo {
    protected Aquarium aquarium;
    public void addAquaticAnimal(AquaticAnimal aquAnimal) {
        this.addAnimal(aquAnimal);
        this.aquarium.add(aquAnimal);
    }
    ...
}
public class Mammal implements Animal { ...}
public class Cetacean extends Mammal implements AquaticAnimal { ... }

```

```

public interface Animal { }
public interface Aquatic { }
public interface AquaticAnimal extends Animal, Aquatic { }
public class Zoo {
    public void addAnimal(Animal animal) { ... }
}
public class Aquarium {
    public void add(Aquatic aquatic) { ... }
}
public class ZooWithAquarium extends Zoo {
    protected Aquarium aquarium;
    public void addAquaticAnimal(AquaticAnimal aquAnimal) {
        this.addAnimal(aquAnimal);
        this.aquarium.add(aquAnimal);
    }
    ...
}
public class Mammal implements Animal { ...}
public class Cetacean extends Mammal implements AquaticAnimal { ... }

public class WaterLily implements Aquatic { ... } // water lily = nénuphar

```

```

public interface Animal { }
public interface Aquatic { }
public interface AquaticAnimal extends Animal, Aquatic { }
public class Zoo {
    public void addAnimal(Animal animal) { ... }
}
public class Aquarium {
    public void add(Aquatic aquatic) { ... }
}
public class ZooWithAquarium extends Zoo {
    protected Aquarium aquarium;
    public void addAquaticAnimal(AquaticAnimal aquAnimal) {
        this.addAnimal(aquAnimal);
        this.aquarium.add(aquAnimal);
    }
    ...
}
public class Mammal implements Animal { ...}
public class Cetacean extends Mammal implements AquaticAnimal { ... }

public class WaterLily implements Aquatic { ... } // water lily = nénuphar
// ... utilisation ...
ZooWithaquarium zoo = new ZooWithAquarium();
Cetacean cet = new Cetacean();
zoo.addAquaticAnimal(cet);
zoo.getAquarium().add(new WaterLily());

```

final

- il est possible d'interdire l'héritage d'une classe
- il suffit de mentionner le modificateur **final** dans l'entête de déclaration de classe.

```
public final class ExtendsNotPossibleClass {  
    ...  
}
```

java.lang.String, java.lang.Boolean, java.net.InetAddress, etc.

- il est possible d'interdire la surcharge (redéfinition) d'une méthode
- il suffit de mentionner le modificateur **final** dans la signature de la méthode.

```
public class SomeClass {  
    public final void someMethod() { ... }  
}  
public class SomeSubClasse extends SomeClass {  
    // ne peut pas redéfinir public void someMethod()  
}
```

```
    dans classe java.applet.Applet  
        public final void setStub(AppletStub stub)  
public static final AudioClip newAudioClip(URL url)
```

final ~ "constant"

Application

Question

Quel(s) intérêt(s) à définir une méthode `final` ?

Application

- fermer le code à l'extension
- garantir le code exécuté indépendamment de tout héritage possible.

Application

- fermer le code à l'extension
- garantir le code exécuté indépendamment de tout héritage possible.

ex : dans une classe `Poll` (*scrutin*), garantir l'anonymat du votant quelque soit le (sous-)type de scrutin :

```
public final void vote(Person person) throws ImpossibleVoteException {  
    // vote and voter (person) are never memorized together  
    if (this.canVote(person)) {  
        Vote vote = person.vote(this.possibleVotes); // gets the vote from person  
        this.storeVote(vote); // stores vote but not the voter  
        this.voteDone(person); // person has voted  
    }  
    else {  
        throw new ImpossibleVoteException ();  
    }  
}  
  
private final void storeVote(Vote vote) {  
    // no information about voter while storing vote  
    this.allVotes.add(vote);  
}
```