
Programmation orientée objet en Java

# Héritage de classes (partie 1)

**Dominique Blouin**
**Télécom Paris, Institut Polytechnique de Paris**
**dominique.blouin@telecom-paris.fr**

# Objectifs d'apprentissage

- **Héritage de classe**
- **Polymorphisme**
- **Liaison dynamique**
- **Mot clé final**

# Rappels

- Les **objets** sont des entités de programme qui communiquent par envois de **messages**.

- Les objets contiennent des valeurs appelées **attributs**. Parmi les attributs, on peut trouver des **références** sur d'autres objets.
  - Une référence sur un objet permet de lui envoyer un **message**.

- Pour chaque type de message que l'objet peut recevoir, l'objet connaît une **méthode** associée au type de message.

- Cette méthode est une procédure qui est **exécutée par l'objet** lorsqu'il reçoit le type de message associé.

# Rappels

- Un type d'objet est décrit par une **classe**.
  - Cette classe décrit les attributs : nom et type de valeur.
  - Elle décrit également les méthodes utilisées pour répondre aux messages.

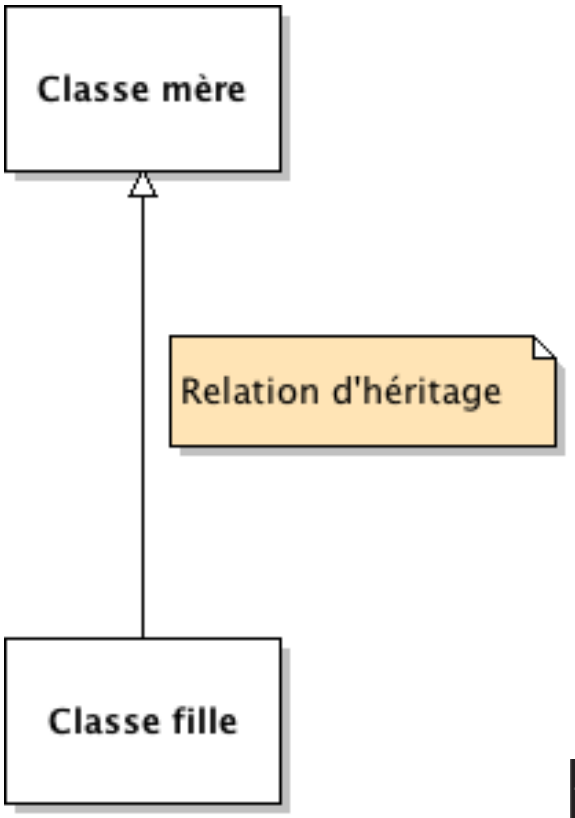
- Le programmeur peut créer des objets à partir de la classe. C'est le processus d'**instanciation**.

- On dit que les objets sont des **instances** de la **classe** ou que les objets **appartiennent** à la **classe**.

# L’héritage de classe

- Une classe **`A`** peut déclarer qu'elle **hérite** d'une classe **`B`**.
  - Cette classe `A` est dite classe **fille** ou **sous-classe** de la classe **`B`**.
  - La classe **`B`** est dite  classe **mère** ou **super-classe** de la classe **`A`**.

- Signification: la classe fille **hérite** des **déclarations** faites dans la classe mère.



# L’héritage de classe

- Lorsque l'on déclare qu’une classe fille hérite d’une classe mère, il est possible d'**enrichir** la classe fille avec des **attributs et des méthodes supplémentaires**.
- On parle alors d'**enrichissement** ou d'**extension modulaire**.
- Il est également possible de **redéfinir** des **méthodes héritées** en donnant une **nouvelle implémentation** de ces méthodes.
- On parle alors de **redéfinition** ou de **substitution**.
- Enrichissement et redéfinition ne sont pas exclusifs.

# Exemple d'héritage et de redéfinition de méthode

```
public class Item {  // Un article dans un magasin

    private double netPrice;

    public double getNetPrice() {
        return netPrice;
    }

    public double getVAT() {      // VAT = Value Added Tax
        return 0.185 * getNetPrice(); // 18,5%
    }

    public double getATIPrice() { // ATI = All Taxes Included
        return getNetPrice() + getVAT();
    }

    ...
}
```

# Exemple d'héritage et de redéfinition de méthode

- Le mot clé **`extends`** permet de déclarer la relation d'héritage :

```
public class LuxuryItem extends Item {

    @Override
    public double getVAT() { // Method overriding
        return 0.33 * getNetPrice(); // 33% tax rate
    }

    ...
}
```

- **`@Override`** est une annotation. Elle est une indication destinée au compilateur pour lui signifier qu'il s'agit d'une **redéfinition de méthode**.
  - Le compilateur vérifiera que c'est bien le cas.

- L'annotation **`@Override`** n'est pas obligatoire mais **fortement recommandée**.

# Enrichissement d'une classe : exemple

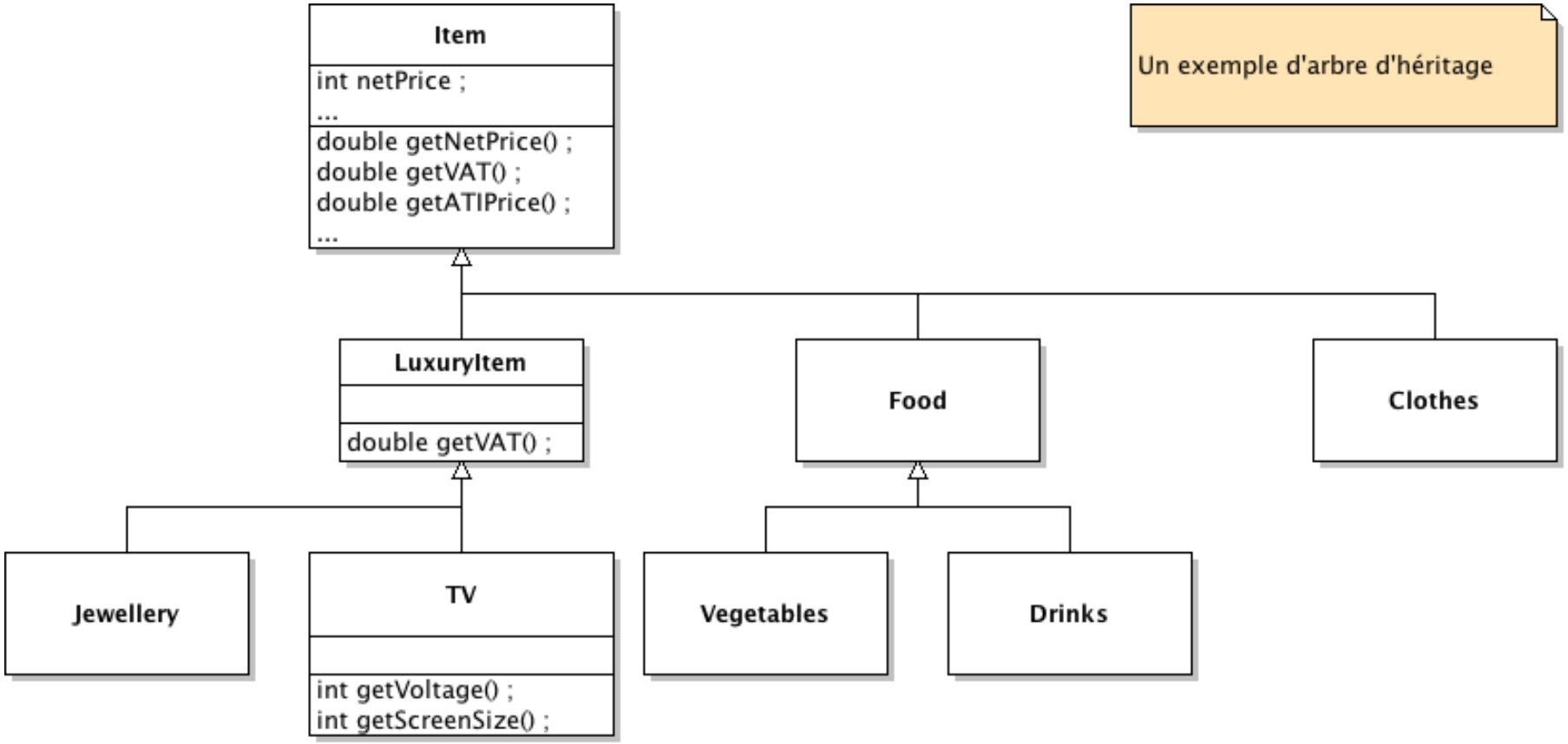
- Un article de luxe particulier :

```
public class Television extends LuxuryItem {

    private int voltage;

    private int screenSize;

    public int getVoltage() {
        return voltage;
    }

    public int getScreenSize() {
        return screenSize;
    }
    ...
}
```

- Les attributs **`voltage`** et **`screenSize`** enrichissent la classe avec des **données supplémentaires**.

# Arbre d'héritage

- On représente la relation d'héritage entre les classes par un arbre dit l'**arbre d'héritage**.



# Racine de l'arbre d'héritage

- Si une classe ne spécifie aucune classe mère, elle hérite par défaut de la classe **`Object`** du package **`java.lang`**.
- Recherche : JAVA SE Object
- En Java, chaque classe ne peut hériter que **d'une seule** classe.
- Dit **héritage simple**.

# Visibilité des éléments de la classe mère

- Quatre types de visibilité (rappel) :
  - **`public`**, **`private`**, **`package`** et **`protected`**.

- Tout ce qui est déclaré **`public`** dans la classe mère est accessible depuis **toutes les classes**.

- Tout ce qui est déclaré **`private`** dans la classe mère n'est accessible **que depuis la classe elle-même**, et non pas depuis les classes filles, ni depuis les classes du même package que celui de la classe mère.

- Tout ce qui est déclaré **`package`** (valeur par défaut) dans la classe mère n'est accessible que depuis les **classes du même package** que celui de la classe mère, incluant les classes filles du même package, mais **pas celles d'un autre package**.

- Tout ce qui est déclaré **`protected`** dans la classe mère n'est accessible que depuis **les classes filles**, indépendamment de leur package, ainsi que depuis les classes du même package, indépendamment de l'héritage.

# Encapsulation des données et utilisation de la visibilité `protected`

- Un attribut déclaré **`protected`** ne respecte pas le principe d'encapsulation.
- En effet, un objet d'une classe fille ou d'une classe du même package peut directement **modifier l'attribut** sans aucun **contrôle**.
- On préfèrera donc utiliser la visibilité **`private`** et on utilisera les accesseurs depuis la classe fille, tout comme pour les autres classes.
- C'est pourquoi on n'utilisera le mot clé **`protected` que pour les méthodes** pour lesquelles on ne veut réserver l'usage qu'aux classes filles ou aux classes du même package.

# Exemple de visibilité des éléments de la classe mère

- Les accesseurs permettent d'accéder aux éléments de visibilité **`private`** :

```
public class Item {

    private double netPrice; // ne pas utiliser protected

    public double getNetPrice() {
        return netPrice;
    }
    ...
}

public class LuxuryItem extends Item {

    @Override
    public double getVAT() {

        // on ne peut pas utiliser netPrice directement
        return 0.33 * getNetPrice(); // 33%
   }
   ...
}
```

# Héritage des constructeurs

- Dans le cas où une classe mère possède des **constructeurs**, les constructeurs de la classe fille doivent impérativement **appeler** l'un des constructeurs de la classe mère.

- L'appel du constructeur de la classe mère doit être la **première instruction** du constructeur de la classe fille.

- Cet appel est réalisé grâce au mot clé **`super`** suivi des paramètres entre parenthèses.

- Le compilateur est vigilant.

# Exemple avec la classe Point

```
public class Point {

    private int xCoord;

    private int yCoord;

    public Point(int xCoord, int yCoord) {
        this.xCoord = xCoord;
        this.yCoord = yCoord;
   }
}

public class ColouredPoint extends Point {

    private Color color;

    public ColouredPoint(int xCoord, int yCoord, Color color) {
        super(xCoord, yCoord); // doit être la premiere  instruction

        this.color = color;
    }
}
```

# Héritage des méthodes et polymorphisme

- Considérons une classe **Shape** qui modélise des formes localisées dans un plan.
  - Une forme est ainsi caractérisée par ses coordonnées dans le plan :

```
public class Shape {

    private int xCoord;
    private int yCoord;

    public Shape(int xCoord, int yCoord) {
        this.xCoord = xCoord;
        this.yCoord = yCoord;
    }

    public int getxCoord() {
        return xCoord;
    }

    public int getyCoord() {
        return yCoord;
    }
}
```

# Une forme rectangulaire

- Un rectangle est une forme particulière ayant une hauteur et une largeur. Elle peut être modélisée par une classe **`Rectangle`** :

```
public class Rectangle extends Shape {

    private int width;
    private int height;

    public Rectangle(int xCoord, int yCoord, int width, int height) {
        super(xCoord, yCoord);

        this.width  = width;
        this.height = height;
    }

    public int getWidth() {
        return width;
    }

    public int getHeight() {
        return height;
    }
}
```

# D’autres formes géométriques…

- Un **carré** est un rectangle particulier ayant une hauteur égale à la largeur.

```
public class Square extends Rectangle {
    public Square(int xCoord, int yCoord, int width) {
        super(xCoord, yCoord, width, width);
    }
}
```

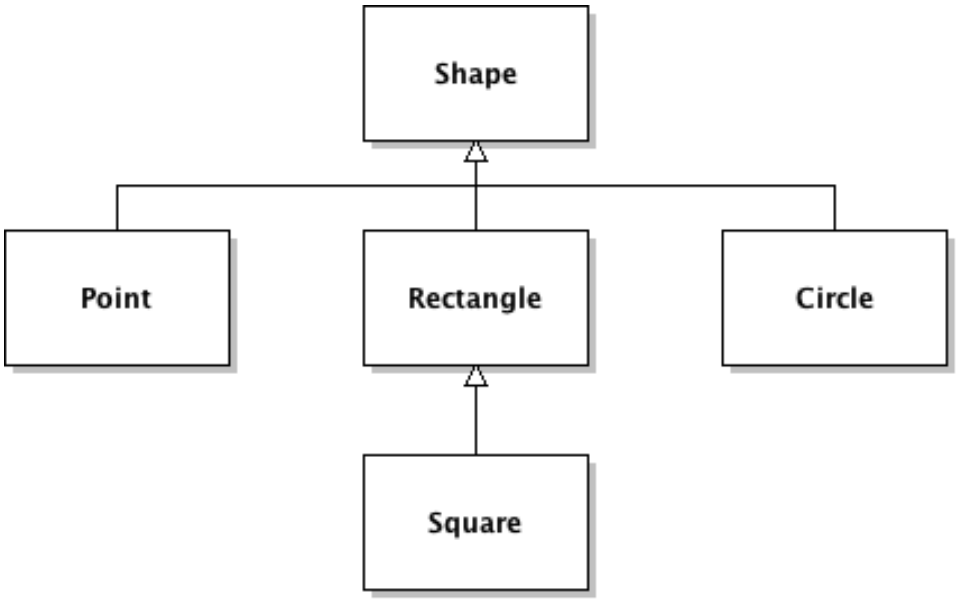
- Un **cercle** est une forme qui a un attribut de **rayon**.

```
public class Circle extends Shape {
   private int radius;
   public Circle(int xCoord, int yCoord, int radius) {
      super(xCoord, yCoord);

      this.radius = radius;
   }
   public int getRadius() {
      return radius;
   }
}
```

- Un **point** est également une forme qui ne nécessite pas d'autres informations que des coordonnées.

```
public class Point extends Shape {
   public Point(int xCoord, int yCoord) {
      super(xCoord, yCoord);
   }
}
```

# Arbre d'héritage




- Techniquement, un objet de la classe **`Square`** possède les trois types **`Square`**, **`Rectangle`** et **`Shape`**.
  - Un objet carré est également un rectangle et il est également une forme.
- De même, un objet de la classe **`Rectangle`** possède les deux types **`Rectangle`** et **`Shape`**. Mais il ne possède pas le type **`Square`**.
- Les objets des classes **`Point`**, **`Square`**, **`Rectangle`** et **`Circle`** ont tous le type **`Shape`**.

# Définition du polymorphisme

- Un objet est toujours instance d'une classe. Ainsi, si nous écrivons :

```
Square square = new Square(10, 10, 100);
```

- L'objet référencé est instance de la classe **`Square`**.
- Mais cet objet a **trois** types: **`Square`**, **`Rectangle`** et **`Shape`**.
- Quand des objets peuvent **avoir plusieurs types**, on parle de **polymorphisme**.

# Exemple d'utilité du polymorphisme : afficher les formes à la console

```
public class Shape {

    private int xCoord;
    private int yCoord;

    public Shape(int xCoord, int yCoord) {
        this.xCoord = xCoord;
        this.yCoord = yCoord;
    }

    public int getXCoord() {
        return xCoord;
    }

    public int getYCoord() {
        return yCoord;
    }

    public print() {
        System.out.print("x = " + getXCoord() + " y = " + getYCoord());
    }
}
```

# Redéfinir l'affichage en fonction du type de forme

- Dans la classe **`Rectangle`**, on utilisera le mot clé **`super`** pour appeler la méthode **`print()`** de la classe mère :

```
@Override
public void print() {
    System.out.print("Rectangle: ");
    super.print();
    System.out.print(" width = " + getWidth() + " height = " + getHeight());
}
```

  - Cela évite de **dupliquer le code** d'affichage des coordonnées de la figure.

- Dans la classe **`Point`** :

```
@Override
public void print() {
    System.out.print("Point: ");
    super.print();
}
```

# Redéfinir l'affichage en fonction du type de forme

- Dans la classe **`Circle`** :

```
@Override
public void print() {
    System.out.print("Circle: ");
    super.print();
    System.out.print(" radius = " + getRadius());
}
```

- Dans la classe **`Square`** :

```
@Override
public void print() {
    System.out.print("Square: ");
    super.print();
    System.out.print(" width = " + getWidth());
}
```

- Est-ce que cette dernière implémentation fonctionne correctement s?

# Une solution possible…

- Dans la classe **Shape** :

```
public void print() {
    printCoordinates();
}

protected void printCoordinates() {
    System.out.print("x = " + getXCoord() + " y = " + getYCoord());
}
```

- Dans la classe **Rectangle** :

```
@Override
public void print() {
    System.out.print("Rectangle: ");
    super.print();
    System.out.print(" width = " + getWidth() + " height = " + getHeight());
}
```

- Dans la classe **Square** :

```
@Override
public void print() {
    System.out.print("Square: ");
    printCoordinates();
    System.out.print(" width = " + getWidth());
}
```

# Afficher à la console une liste d'éléments de différents types de formes

```
ArrayList<Shape> shapes = new ArrayList<Shape>();
shapes.add(new Square(0, 0, 10));
shapes.add(new Rectangle(0, 0, 10, 20));
shapes.add(new Circle(0, 0, 10));
shapes.add(new Point(0, 0));

// Printing the shapes
for (Shape shape : shapes) {
    shape.print();
}
```

- Question : pour chaque figure de la liste, quelle sera la **méthode appelée** lors de l'affichage?

# Liaison de méthode

- Deux stratégies sont possibles :

- Recherche **statique** de méthode : le compilateur considère que le type déclaré de la variable **`shape`** est **`Shape`** et il décide que c'est la méthode **`print()`** de la classe **`Shape`** qui doit être exécutée.
  - Le mot **statique** signifie que la méthode qui sera exécutée est déterminée **au moment de la compilation**.
  - On parle alors de liaison **statique**.

- Recherche **dynamique** de méthode : le compilateur programme une recherche de la méthode basée sur la **classe d'instanciation** de l'objet référencé qui sera examinée **au moment de l'exécution**.
  - Le mot **dynamique** signifie que la méthode exécutée ne sera déterminée qu'**au moment de l'exécution**.
  - On parle alors de liaison **dynamique**.

- Java pratique la recherche **dynamique** de méthode.
  - C'est un langage dit de **liaison dynamique**.

# Liaison dynamique de méthode

- La méthode exécutée lors de l'appel **`shape.print()`** est déterminée au moment de l'exécution du programme.
- Java regarde quelle est la **classe d'instanciation** de l'objet référencé par la variable **`shape`** et choisit la méthode **`print()`** de cette classe.
- Ce mécanisme, fort bien implémenté, ne coûte pas très cher en temps d'exécution.

```
ArrayList<Shape> shapes = ...;

for (Shape shape : shapes) {
    shape.print();
}
```

- Comment cela est-il réalisé?
  - Recherche : JAVA SE Object
- Examinez cette méthode de la classe **`Object`** :

```
public final Class<?> getClass()
```

# La méthode `toString()`

- Dans les TP précédents, vous avez **redéfini** la méthode **`toString()`** dans votre classe **`Robot`**.

- Que se passe-t-il lorsque vous appelez :

```
System.out.print(myRobot);
```

- Dans la classe de l'attribut statique **`System.out`** (qui est de la classe **`PrintStream`**), l'affichage appelle simplement la méthode **`toString()`** de la classe **`Object`**.

- La lyaison dynamique détermine alors que c'est la méthode de votre classe **`Robot`** qui doit être appelée.

# Le mot clé `final`

- Même si le mécanisme de liaison dynamique est très efficace, il a quand même un coût.

- Il existe toutefois des cas où il est possible de se passer de la recherche dynamique de méthode : lorsque la méthode à exécuter peut être déterminée **statiquement** au moment de la **compilation**.

- Pour cela, on déclare qu'une méthode est **`final`**, ce qui veut dire qu'il est interdit qu'une sous-classe **redéfinisse** cette méthode.

- Ainsi, puisque la méthode ne sera jamais redéfinie, son appel peut être déterminé **statiquement** au moment de la compilation sans qu'il y ait ambiguïté sur la méthode à appeler.

# Exemple

- Rajoutons une méthode dans la classe **`Shape`** :

```
public void println() {
    print();
    System.out.println();
}
```

- Cette méthode n'a pas besoin d'être redéfinie dans les sous-classes ; elle est correcte pour **tous les types** de figure.
  - Pourquoi?

- Donc, pour un appel **`shape.println()`**, il est inutile de pratiquer la recherche dynamique de méthode car la méthode qui sera exécutée est connue, ce sera toujours celle (unique) de la classe **`Shape`**.

- Le programmeur le sait, mais le compilateur lui n'a aucun moyen de le savoir.
  - Quand il compile la classe **`Shape`**, il ne peut pas savoir si la méthode **`println()`** sera redéfinie ou pas car il ne connaît pas toutes les sous-classes.

# Exemple

- C'est donc au **programmeur** d'indiquer grâce au mot clé **`final`** que cette méthode ne sera pas redéfinie :

```
public final void println() {
    print();
    System.out.println();
}
```

- Grâce à cette déclaration, le compilateur **sait** que la méthode ne sera jamais redéfinie par **aucune** des sous-classes, car cela générerait une **erreur** à la compilation.

- Un bon compilateur évitera donc de programmer une recherche dynamique dans ce cas.

- L'utilisation de mot clé **`final`** pour les méthodes qui ne seront jamais redéfinies permet au compilateur de réaliser des **optimisations** qui améliorent la vitesse d'exécution des programmes.

# Autres usages du mot clé `final` : classe finale

- Le mot clé **`final`** peut également qualifier une **classe**.
  - C'est le cas de la classe `String` du JDK :

```
public final class String {
    ...
}
```

- Il n'est **pas possible** d'hériter d'une classe déclarée **`final`**.
  - Les méthodes d'une classe déclarée **`final`** sont implicitement toutes déclarées **`final`** également.

- Les différents usages du mot clé **`final`** permettent au compilateur de réaliser des optimisations :
  - classe finale.
  - méthode finale.
  - attribut final.
  - variable de classe finale.

- Rendre une classe finale peut également **renforcer la sécurité**.
  - Par exemple, la classe **`String`** est vitale car elle est utilisée par le compilateur et par d'autres parties importantes de Java.

- Il est donc important **d'interdire de changer le comportement des méthodes** de la classe, ce qui pourrait se faire en redéfinissant la classe.

# Autres usages du mot clé `final` : mutabilité des objets

- Le mot clé **`final`** peut également qualifier les **attributs** d'une classe.
- Les valeurs de ceux-ci ne pourront alors être affectés qu'**une seule fois**.
- Ce sont des attributs **non mutables**.
  - On ne pourra faire changer la position de la forme qu'en **instanciant un nouvel objet**.
- Qu'en est-il si la forme est un robot du simulateur?

```
public class Shape {

    private final int xCoord;

    private final int yCoord;

    public Shape(int xCoord, int yCoord) {
        this.xCoord = xCoord;
        this.yCoord = yCoord;
    }

    public int getXCoord() {
        return xCoord;
    }

    public int getYCoord() {
        return yCoord;
    }
```

# Autres usages du mot clé `final` : paramètres et variables finales

- Le mot clé **`final`** peut également qualifier les **paramètres** d'une méthode.
- Il peut également qualifier les **variables locales** d'une méthode.

```
public void doSomething(final int xPar) {
    ...
    final int number = ... ;
    ...
}
```

- Le compilateur est parfaitement capable de déterminer si le programme modifie une variable ou un paramètre.
- La qualification en **`final`** ne sert alors qu'à indiquer que le programmeur veut être certain que la variable ou le paramètre ne sera pas modifié.
- Dans les deux cas, le compilateur vérifiera qu'aucune instruction intempestive ne modifie la valeur du paramètre ou de la variable.

# Héritage simple et multiple

- L'héritage est dit **simple** si chaque classe hérite **au plus** d'une classe.
  - Par exemple **Java**, SmallTalk et Ada.
  - L'héritage est alors matérialisé par un **arbre** ou une **forêt d'arbres**.

- L'héritage est dit **multiple** si une classe peut hériter de **plusieurs** classes.
  - Par exemple C++ et Eiffel.
  - L'héritage est alors matérialisé par un ou plusieurs **graphes orientés**.