## Héritage

- L'héritage est un mécanisme fondamental de réutilisation du code : l'idée est de cerner les similitudes et de les réunir dans une *classe de base*.
- Une *classe dérivée* est une classe qui hérite des membres non privés d'une classe de base. On dit aussi que la classe dérivée est une *spécialisation* de la classe de base ou que c'est une *sous-classe*. La classe de base est une *super-classe* de la classe dérivée.
- Si la classe dérivée ne *redéfinit* pas une méthode de sa super-classe, c'est la méthode de la super-classe qui sera appelée. Sinon, ce sera la méthode redéfinie (voir plus loin).

### Remarque

- Ce mécanisme produit une *arborescence d'héritage* dont la racine est la classe `java.lang.Object`. Même si une classe `A` ne peut dériver *directement* que d'une seule classe de base `B`, cette dernière peut elle-même hériter d'une autre classe `C`, etc.
- On peut donc dire que `A` hérite indirectement de `C`. Selon ce principe, toute classe Java hérite donc directement ou indirectement de `java.lang.Object`.



## Héritage

La difficulté consiste évidemment à cerner les similitudes entre les classes le plus tôt possible dans le cycle de développement. Une autre difficulté consiste à définir les interactions des différents objets.
Voici quelques points qui permettront de confirmer les intuitions :

- un technicien *est un* employé, un rectangle *est une* figure géométrique, une socket *est un* point de communication.
- un technicien *a un* nom, un rectangle *a une* couleur de contour, une socket *a un* nombre d'octets envoyés depuis sa création.
- un technicien *peut être* évalué, un rectangle *peut être* agrandi, une socket *peut être* utilisée pour envoyer un flot d'octets.

### Remarque

- Attention à ne pas utiliser l'héritage à mauvais escient. Une classe `B` ne doit hériter d'une classe `A` que si les objets `B` *sont aussi* des objets A.
- Ne pas confondre héritage (*« est un(e) »*) et composition (*« a un(e) »*) !

## Héritage

Pour indiquer que la classe `Technicien` dérive (ou *hérite*) de la classe `Employé`, on utilise la syntaxe suivante :

```
class Technicien extends Employé { ... }
```

Ici, la classe `Technicien` hérite de tous les membres (attributs et méthodes) public de `Employé`, ainsi que de ses membres protégés (et à visibilité par défaut si elle est définie dans le même paquetage).

### Héritage simple ou multiple

*En Java, comme en C# et comme en Ruby, une classe ne peut dériver que d'une seule classe de base : il n'y a pas d'héritage multiple comme en C++ mais cela est compensé en partie par le concept d'interface car une classe peut implémenter plusieurs interfaces.*



## Héritage

Un constructeur d'une classe dérivée doit toujours appeler un constructeur de sa classe de base.

- Cet appel peut être *implicite* : ce sera le constructeur *par défaut* de la classe de base qui sera appelé. Si la classe de base n'a pas de constructeur par défaut, cela provoquera une erreur.
- Cet appel peut être *explicite* en utilisant un appel `super(...)` : ce sera le constructeur qui a ce prototype dans la classe de base qui sera appelé.

```
class Employé {
   private String nom;
   private short age;
   public Employé(String nom, short age) {
       this.nom = nom;
       this.age = age;
   }
   public Employé() {
     this("n/a", 0);
   }
} // class Employé

class Technicien extends Employé {
    private String compétences;
    public Technicien(String nom, short age, String compétences) {
      super(nom, age);
      this.compétences = compétences;
    }
} // class Technicien
```

## Héritage

- L'appel à `super(...)` doit être la première instruction du constructeur de la classe dérivée.
- Si un constructeur n'utilise pas `super(...)` comme première instruction, Java insère automatiquement `super()` afin d'appeler le constructeur par défaut de la super-classe. L'exception à cette règle est que lorsque la première instruction d'un constructeur est `this(...)`, Java n'insèrera rien (on considère que c'est l'autre constructeur qui doit effectuer cet appel).
- `super` comme simple mot-clé (sans parenthèses) permet également de désigner des méthodes de la classe parente lorsque celles-ci ont été redéfinies :

```
class A {
   ...
   @Override      // Redéfinition de Object.toString()
   public String toString() {...}
} // class A

class B extends A {
   ...
   @Override      // Redéfinition de A.toString()
   public String toString() { /* ici on peut appeler super.toString() pour désigner la méthode toString() de A */ }
} // class B
```



## Exemple

```
// Classe pour réprésenter un point du plan
class Point {
  private int x, y;  // Coordonnées du point

  public Point(int x, int y) { this.x = x; this.y = y; }
  public Point()             { this(0, 0); }

  public int getX() { return x; }
  public int getY() { return y; }
  public void deplace(int deltaX, int deltaY) {
    x += deltaX;
    y += deltaY;
  }
  @Override
  public String toString() { return String.format("(%d, %d)", x, y); }
} // class Point

// Classe permettant d'étiqueter un point du plan
class PointNommé extends Point {
    private String nom;

    public PointNommé (String n, int x, int y){
        super(x, y);   // appel de Point(x, y)
        nom = n;
    }
    public PointNommé (String n) { this(n, 0, 0); }

    public String getNom()        { return nom; }
    public void setNom(String n)  { nom = n; }
    @Override
    public String toString()      { return nom + " " + super.toString(); }
} // class PointNommé
```

## Exemple (suite...)

```
// Classe de test
class TestPoint {
    public static void main(String[] args) {
      Point p1 = new PointNommé("Point 1");
      PointNommé p2 = new PointNommé("Point 2", 10, 15);

      p1.deplace(5, 20);
      System.out.printf("p1 : %s%np2 : %s%n", p1, p2);
    }
}
```

### Commentaires sur l'exemple

- La classe `PointNommé` hérite de la classe `Point`, qui est sa super-classe : un objet `PointNommé` *est* un `Point`.
- La classe `PointNommé` hérite donc de toutes les méthodes d'instance non-`private` de la classe `Point` et peut accéder directement à ses attributs non privés.
- La classe `PointNommé` peut ajouter ses propres attributs et méthodes (extension de la classe `Point`).
- Si `PointNommé` définit une méthode existant déjà *avec la même signature*, on dit que cette méthode est *redéfinie* (à ne pas confondre avec la *surcharge*). Ici, on redéfinit `toString()` et on l'indique avec l'annotation `@Override` (notez que `Point` redéfinit la méthode `toString()` de `Object`).
- Si `PointNommé` définit un attribut de même nom qu'un attribut de sa super-classe, celui-ci est *masqué* : on utilisera `super.attribut` pour le désigner.



## Chaînage des constructeurs et des finaliseurs

- Comme l'appel au constructeur de la super-classe est le premier appel de chaque constructeur, on a un *chaînage* des appels des constructeurs de la hiérarchie en commençant par celui de `Object` : pour construire un `PointNommé`, il faut d'abord construire un `Point` et pour construire un `Point`, il faut d'abord construire un `Object`.
- Par contre, le finaliseur d'un objet *n'appelle jamais* implicitement celui de sa super-classe.
- S'il le faut, on appellera *explicitement* le finaliseur de la super-classe en effectuant un appel à `super.finalize()` comme *dernière* instruction du finaliseur de la classe dérivée.

## Méthodes virtuelles et polymorphisme

Rappel du problème :

- On peut créer des instances de classes dérivées d'une classe de base.
- On veut leur appliquer un traitement *défini dans la classe de base*.
- Mais ce traitement dépend de la classe dérivée :
  - on veut obtenir une description appropriée pour chaque employé (le traitement de base consiste à obtenir une description d'un employé, quelle que soit sa catégorie).
  - on veut dessiner toutes les figures géométriques (le traitement de base consiste à dessiner une figure, quel que soit son type).
  - on veut envoyer des données en passant par un point de communication (le traitement de base consiste à envoyer des données, quel que soit le protocole de communication).



## Méthodes virtuelles et polymorphisme

Solution :

- Ce que l'on a appelé traitement de base est réalisé par une *méthode virtuelle*, qui est définie dans la classe de base et *redéfinie* dans la classe dérivée (même nom, mêmes paramètres, même type de résultat).
- On a donc une méthode qui a plusieurs corps : lors de l'appel à cette méthode *sur une référence du type de la classe de base*, le corps exécuté est choisi au moment de l'exécution du programme. Le fait qu'une méthode puisse prendre plusieurs corps différents en fonction du contexte se traduit par le terme de *polymorphisme*.
- Une classe dérivée n'est pas obligée de redéfinir une méthode virtuelle de sa classe de base. Une classe de base qui contient au moins une méthode virtuelle est une *classe polymorphe*.

# Méthodes virtuelles et polymorphisme

## Remarques

- En Java, les méthodes sont virtuelles par défaut : les mots-clés `virtual` et `override` n'existent donc pas. Pour qu'une méthode ne puisse pas être redéfinie (ce qui correspond au comportement par défaut de C++ et C#), il faut l'indiquer explicitement avec le mot-clé `final`.
- Une *redéfinition* est différente d'une *surcharge* : la surcharge consiste à définir plusieurs méthodes de même nom dans la même classe, mais avec des paramètres différents.
- Les méthodes de classe et les méthodes privées ne peuvent évidemment pas être redéfinies dans une classe dérivée.
- Pour éviter les erreurs de redéfinition, il est conseillé de précéder chaque redéfinition de méthode de l'annotation `@Override`.



# Exemple de redéfinition de méthode

```
class Employe {
   protected String nom;
   public Employe(String nom) { this.nom = nom; }
   public void description() {  // Traitement de base
      System.out.printf("Nom : %s%n", nom);
   }
}
class Secretaire extends Employe {
   public Secretaire(String nom) { super(nom); }
   @Override
   public void description() {
     super.description();
     System.out.println("  Fonction : Secrétaire");
   }
}
class Technicien extends Employe {
   public Technicien(String nom)  { super(nom); }
   @Override
   public void description() {
     super.description();
     System.out.println("  Fonction : Technicien");
   }
}
public final class Redefinitions {
   public static void main(String[] args) {
      Employe[] employes = new Employe[3];
      employes[0] = new Technicien("Joe");
      employes[1] = new Secretaire("Jack");
      employes[2] = new Secretaire("William");
      for (Employe employe : employes) {
          employe.description();
      }
    }
}
```

## Exemple de redéfinition de méthode (suite)

Ce programme produira le résultat suivant :

```
Nom : Joe
  Fonction : Technicien
Nom : Jack
  Fonction : Secrétaire
Nom : William
  Fonction : Secrétaire
```

**Remarque**

*Généralement, les méthodes virtuelles redéfinies appellent à un moment ou à un autre la méthode correspondante dans la classe de base. Sinon, c'est qu'il y a probablement un problème de conception.*



## Polymorphisme et liaison différée

- Un objet a un type *apparent* (ou *statique*) : celui qu'il a au moment de la compilation (donc celui de sa référence).
- Il a un type *réel* (ou *dynamique*) : celui qu'il a au moment de son utilisation.
- Quand une méthode est appliquée à un objet, c'est toujours celle associée au *type dynamique* qui est choisie.
- Le compilateur ne connaît que le *type statique* : pour chaque méthode, il ajoute donc un code effectuant une recherche dynamique de méthode (*dynamic method lookup*) afin de réaliser une *liaison différée* (*late binding*) au moment de l'exécution.
- La recherche d'une méthode redéfinie est donc moins rapide qu'un appel direct...

# Polymorphisme et liaison différée

## Illustration

```
Point p = new PointNomme("A", 1, 2);  // Un PointNomme EST UN Point => upcasting
System.out.println(p);          // appel de toString() sur p
System.out.println(p.getNom()); // Erreur : un Point n'a pas de getNom()
```

## Commentaires

- **À la compilation**, il doit exister une méthode `toString()` et `getNom()` dans la classe `Point`, ou l'une de ses ancêtres (sinon : erreur de compilation).
- **À l'exécution**, l'interpréteur recherche la méthode `toString()` à appeler : c'est la dernière en partant du type apparent (`Point`) et en allant vers le type dynamique (`PointNomme`). C'est ce que l'on appelle un « *dispatch simple* ».
- Ce comportement peut sembler étrange, voire surprenant puisque, ici, cela aurait un sens d'appeler `getNom()` sur `p`... Le problème est que le compilateur ne le sait pas.