

# Introduction à Java

Cours 3: Programmation Orientée Objet en Java

Stéphane Airiau

Université Paris-Dauphine

## But : ne pas coder la même chose

```
1 public class Personnage {
2     public String nom;
3
4     // constructeur par défaut
5     public Personnage () {
6         nom = "Inconnu";
7     }
8
9     public Personnage (String name) {
10        nom = name;
11    }
12 }
```

On veut maintenant faire des classes `Gaulois` et `Romain` pour avoir des comportements plus spécifiques.

Comment s'y prendre ?

## But : ne pas coder la même chose

```
1 public class Personnage {
2     public String nom;
3
4     // constructeur par défaut
5     public Personnage () {
6         nom = "Inconnu";
7     }
8
9     public Personnage (String name) {
10        nom = name;
11    }
12 }
```

On veut maintenant faire des classes `Gaulois` et `Romain` pour avoir des comportements plus spécifiques.

Comment s'y prendre ?

➡ recopier ce qui a été fait dans la classe `Personnage` et ajouter des méthodes spécifiques.

## But : ne pas coder la même chose

```
1 public class Personnage {
2     public String nom;
3
4     // constructeur par défaut
5     public Personnage () {
6         nom = "Inconnu";
7     }
8
9     public Personnage (String name) {
10        nom = name;
11    }
12 }
```

On veut maintenant faire des classes `Gaulois` et `Romain` pour avoir des comportements plus spécifiques.

Comment s'y prendre ?

➡ ~~recopier ce qui a été fait dans la classe `Personnage` et ajouter des méthodes spécifiques.~~

✘ On ne veut pas dupliquer de code !

## But : ne pas coder la même chose

```
1 public class Personnage {
2     public String nom;
3
4     // constructeur par défaut
5     public Personnage () {
6         nom = "Inconnu";
7     }
8
9     public Personnage (String name) {
10        nom = name;
11    }
12 }
```

On veut maintenant faire des classes `Gaulois` et `Romain` pour avoir des comportements plus spécifiques.

Comment s'y prendre ?

➡ ~~recopier ce qui a été fait dans la classe `Personnage` et ajouter des méthodes spécifiques.~~

✘ On ne veut pas dupliquer de code !

Java propose l'héritage comme solution.

## Héritage

---

L'héritage permet à un objet d'acquérir les propriétés d'un autre objet ➡  
factorisation des connaissances :

- la classe **mère** (ou classe **de base**) est plus générale
- ➡ elle contient les propriétés communes à toutes les classes **filles** (ou classes **dérivées** ou **héritées**).
- Les classes filles ont des propriétés plus spécifiques.
- ➡ On obtient une hiérarchie de classes.

Pour exprimer qu'une classe est une classe fille, on utilise le mot-clé **extends** dans la déclaration d'une classe :

```
1 | class <nom classe fille> extends <nom classe mère>
```

En Java, on hérite d'une **seule** et **unique** classe.

(une classe qui n'hérite d'aucune classe héritent en fait implicitement de la classe Object)

## Conséquences

---

- Que se passe-t-il pour les variables d'instance
- Que se passe-t-il pour les méthodes de la classe mère
- Constructeurs

### Les méthodes ou attributs

- **public** sont toujours accessibles par une classe fille (bien sûr !)
- **private** restent inaccessibles, même pour une classe fille. Evidemment, les attributs privés sont tout de même hérités : même si on n'a pas directement accès aux attributs, ils sont bien présents !
- ➡ nouvelle portée **protected** : seules la classe et les classes dérivées ont accès à des membres déclarés **protected**.



Pour les méthodes **public** ou **protected**, on a le choix :

- soit le comportement est le même : on peut/doit omettre la ré-écriture de la méthode
- soit le comportement est différent : on peut ré-écrire la méthode

il existe deux références pour parcourir la hiérarchie :

- **this** : est une référence sur l'instance de la classe.
- **super** : est une référence sur l'instance mère.

Evidemment, on peut ajouter des méthodes spécifiques à la classe fille !

- 1 appeler le constructeur de la classe mère : la méthode se nomme **super** tout simplement.
  - si l'appel n'est pas explicite, Java va essayer d'appeler automatiquement le constructeur par défaut (sans argument).
  - ☞ si vous n'avez pas défini de constructeur par défaut dans la classe mère et que vous avez un autre constructeur dans la classe mère, il faudra appeler le constructeur de la classe mère de façon explicite
- 2 faire des traitements spécifiques pour la classe fille.

## Exemple

```
1 public class Personnage {
2     private String nom;
3     // Constructeur
4     public Personnage (String name) {
5         this.nom = name;
6     }
7
8     public String presentation() {
9         return "Je m'appelle" + name;
10    }
11 }
```

```
1 public class Gaulois extends Personnage {
2
3     public Gaulois (String name) {
4         super (name);
5     }
6
7     public String presentation() {
8         return super.presentation() + " je suis un gaulois";
9     }
10 }
11
12 public static void main (String [] args) {
13     Gaulois asterix = new Gaulois ("Astérix");
14     System.out.println ( asterix.presentation());
15 }
```

## Opérateur `instanceof`

Opérateur pour vérifier si une classe est bien un instance d'une classe.

```
1 | public class Personnage { ... }
```

```
1 | public class Gaulois extends Personnage { ... }
```

```
1 | public class IrreductibleGaulois extends Gaulois { ... }
```

```
1 | public class Romain extends Personnage { ... }
2 | ...
5 |     public static void main(String[] args) {
6 |         IrreductibleGaulois asterix = new IrreductibleGaulois();
7 |         System.out.println( asterix instanceof Personnage);
8 |         System.out.println( asterix instanceof Gaulois);
9 |         System.out.println( asterix instanceof Romain);
```

## Opérateur `instanceof`

Opérateur pour vérifier si une classe est bien un instance d'une classe.

```
1 | public class Personnage { ... }
```

```
1 | public class Gaulois extends Personnage { ... }
```

```
1 | public class IrreductibleGaulois extends Gaulois { ... }
```

```
1 | public class Romain extends Personnage { ... }
2 | ...
5 | public static void main(String[] args) {
6 |     IrreductibleGaulois asterix = new IrreductibleGaulois();
7 |     System.out.println( asterix instanceof Personnage);
8 |     System.out.println( asterix instanceof Gaulois);
9 |     System.out.println( asterix instanceof Romain);
```

true

true

false

Astérix est bien un Gaulois, c'est même un irréductible Gaulois, et surtout pas un Romain.

## Polymorphisme

---

L'exemple précédent montre qu'un objet peut avoir **plusieurs** types. C'est ce que l'on appelle le **polymorphisme**.

Le polymorphisme et le transtypage implicite nous permettent de manipuler des objets qui sont issus de classes différentes, mais qui partagent un même type.

```
1 | Personnage asterix = new Gaulois("Astérix");
```

```
1 | Gaulois obelix = new Gaulois("Obélix");  
2 | Gaulois asterix = new Gaulois("Astérix");  
3 | Personnage cleopatre = new Personnage("Cléopâtre");  
3 | Personnage[] distribution = new Personnage[3];  
4 | distribution[0] = asterix;  
5 | distribution[1] = obelix;  
6 | distribution[2] = cleopatre;
```

# Polymorphisme

---

```
1 | Personnage asterix = new Gaulois("Astérix");
```

Dans l'exemple `asterix` est déclaré comme un `Personnage`, même si l'objet est en fait un `Gaulois`.

Comme la variable est déclarée comme un `Personnage`, on ne peut pas appeler une méthode spécifique d'une classe dérivée comme `Gaulois`.

Par exemple :

`asterix.avoirPeurQueLeCielTombeSurMaTete()` ; n'est **pas** permis !

⇒ **si un objet  $o$  est déclaré avec un type  $T$ , on ne peut appeler que des méthodes du types  $T$  sur l'objet  $o$  !**

# Recherche dynamique d'un membre

Les trois classes possèdent une méthode `presentation()`

Java choisit la méthode appropriée au moment de l'**exécution**.

⇒ on a une liaison dynamique.

Au moment de la compilation, on va vérifier si la méthode appliquée à un `Personnage` est bien une méthode de la classe `Personnage` ou de ses parentes.

⇒ on ne pourrait pas appeler `asterix.frappeRomains()`

⇒ si un objet `o` est déclaré avec un type `T`, on ne peut appeler que des méthodes du types `T` sur l'objet `o`!

**Mais** la méthode exécutée est celle qui correspond au type le plus spécifique de l'objet `o`.

```
1 public class Personnage {
2     ...
3     public String presentation() {
4         return "je m'appelle "+nom;
5     }
6 }
```

```
1 public class Gaulois extends Personnage {
2     public Gaulois (String name) { super (name); }
4     @Override public String presentation() {
5         return super.presentation() + "je suis un gaulois";
6     } }
```

```
1 public class IrreductibleGaulois extends Personnage {
2
3     public IrreductibleGaulois (String name) { super (name); }
4     @Override public String presentation() {
5         return super().presentation()
6             + " et je ne crains pas les romains";
7     }
8
9     public void frappeRomains () {
10        System.out.println("qu'est-ce qu'on s'amuse!"); }
11 }
12 public static void main (String[] args) {
13     Random generator = new Random ();
14     Personnage mystere;
15     if (generator.nextBoolean ())
16         mystere = new Gaulois ("Astérix");
17     else
18         mystere = new Romain ("Jules");
19     System.out.println(mystere.presentation());
20 }
```



## Le mot-clé **final**

---

- pour une classe : une classe **final** n'aura pas de classe fille
  - ↳ raison de sécurité pour éviter des « détournements ».
  - exemple : classe `String`
- pour une méthode : cette méthode ne pourra pas être re-définie dans une classe dérivée
- pour une variable : la variable ne pourra être modifiée.

## Tout objet hérite de la classe `Object`

---

Modifier and Type	Method Description
<code>protected Object</code>	<code>clone()</code> Creates and returns a copy of this object.
<code>boolean</code>	<code>equals(Object obj)</code> Indicates whether some other object is "equal to" this one.
<code>protected void</code>	<code>finalize()</code> Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
<code>Class&lt;?&gt;</code>	<code>getClass()</code> Returns the runtime class of this <code>Object</code> .
<code>int</code>	<code>hashCode()</code> Returns a hash code value for the object.
<code>String</code>	<code>toString()</code> Returns a string representation of the object.

## Classes et méthodes abstraites

---

Contexte : A bien y réfléchir, on n'utilisera jamais un objet de la classe `Personnage`, on utilisera toujours un objet d'une classe dérivée (`Romain`, `Gaulois`, `Animaux`, etc).

Pour certaines méthodes, on utilisera toujours la méthode de la classe dérivée, l'implémentation dans la classe `Personnage` est inutile.

Cependant, on veut forcer l'implémentation de ces méthodes dans la classe dérivée.

Solution : utiliser une méthode **abstraite** (mot-clé **abstract**)

- une méthode **abstraite**
  - n'a pas de corps
  - doit être implémentée dans les classes dérivées
- une classe abstraite
  - est une classe qui contient une méthode abstraite
  - ne peut pas être instanciée

## Exemple classe abstraite

---

```
1 public abstract class Personnage {
2
3     String nom;
4
5     public Personnage (String name) ;
6
7     // à définir dans les classes filles
8     public abstract void presentation () ;
9
10    // partagée par toutes les classes dérivées
11    public void jeMappelle () {
12        System.out.println(" je m' appelle " + nom) ;
13    }
14 }
```

N.B. Même si `Personnage` est abstraite, on peut cependant avoir un constructeur :

- par exemple si on veut initialiser des attributs avant d'initialiser l'objet (par exemple des attributs **final**)

## Interfaces

---

En Java, on a un héritage **simple** : on ne peut hériter que d'une seule classe.

Les interfaces offrent un mécanisme pour réaliser de l'héritage **multiple**.

Une interface est une sorte de standard

- pour suivre le standard, une classe doit posséder les méthodes et les constantes déclarées dans l'interface.
- ➡ on dit que la classe implémente l'interface.
- Une classe peut implémenter **plusieurs** interfaces.

```
1 [public] interface <nom interface>
2     [extends <nom interface 1> <nom interface 2> ... ] {
3     // méthodes ou des attributs static
4 }
```

# Interfaces

---

- Toute méthode déclarée dans une interface est **abstraite**
- Les méthodes sont implicitement déclarées comme telles (i.e. il n'est pas nécessaire d'ajouter le mot-clé `abstract`)
- Tout attribut est implicitement déclaré `static` et `final`.

```
1 public interface Combattant {
2     public void attaque(Personnage p);
3     public void defend(Combattant c);
4 }
```

```
1 public class IrreductibleGaulois implements Combattant {
2     ...
3     public void attaque(Personnage p) {
4         gourdePotionMagique.bois();
5         while (p.isDebout())
6             coupsDePoing(p);
7     }
8
9     public void defend(Combattant c) {
10        esquive();
11        attaque(c);
12    }
13 }
```

- Portée

- **public** : visible de tous
- **private** : visible seulement depuis la classe
- **protected** : visible des classes filles

⇒ par défaut, choisir plutôt **private**

- **static** : pour la définition d'une variable de classe
- **final** : pour un attribut non modifiable
- type de l'attribut

⇒ déclaration :

[portée] [**static**] [**final**] <type> nomAttribut

- Portée

- **public** : callable depuis n'importe quelle classe
- **private** : callable seulement à l'intérieur de la classe
- **protected** : callable à l'intérieur de la classe et à l'intérieur des classes de la descendance.

- **static** : pour la définition d'une méthode de classe

- **final** : pour une méthode qui ne pourra pas être redéfinie par une classe fille

- type de retour (pour une fonction) ou **void** pour une procédure

- nom de la méthode

- liste des arguments

➡ on peut avoir des méthodes avec le même nom et le même type de retour tant qu'elles ont des listes d'arguments différentes !

➡ on parle de **surchage**