Introduction à Java

Cours 3: Programmation Orientée Objet en `Java`

Stéphane Airiau

Université Paris-Dauphine

But : ne pas coder la même chose

```java
public class Personnage {
   public String nom;

   // constructeur par défaut
   public Personnage(){
     nom = "Inconnu";
   }

   public Personnage(String name){
     nom = name;
   }
}
```

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

```java
public class Personnage {
   public String nom;

   // constructeur par défaut
   public Personnage(){
     nom = "Inconnu";
   }

   public Personnage(String name){
     nom = name;
   }
}
```

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 hierarchie 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`)

Exemple

```java
public class Personnage {
   public String nom;

   // constructeur par défaut
   public Personnage(){
     nom = "Inconnu";
   }
}
   public String presentation(){
     return "mon nom est" + nom;
   }
}
```

```java
public class Gaulois extends Personnage {


   // constructeur par défaut
   public Gaulois(){
     Que doit-on écrire ?
   }

   public String presentation(){
     Que doit-on écrire ?
   }
}
```

Conséquences

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

Membres protégés – `protected`

Les méthodes ou attributs

- `public` sont toujours accessibles par une classe fille
  (bien sûr !)
- `private` restent inaccessibles, même pour une classe fille.

  Même si vous n'y avez pas directement accès, les variables et méthodes `private` existent bien pour une instance d'une classe fille, elles sont simplement cachées.

⇨ nouvelle portée `protected` : seules la classe et les classes dérivées ont accès à des membres déclarés `protected`.

méthodes de la classe mère

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

  On peut utiliser une annotation `@Override` pour souligner que l'on rédéfinit une méthode de la classe mère.

  ➪ `Java` vérifiera si on effectue vraiment une redéfinition !

il existe deux références pour parcourir la hierarchie :

- **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 !

Constructeur de la classe fille

La signature du constructeur suit la règle habituelle.
Pour l'implémentation, il y a deux étapes :

1. appeler le constructeur de la classe mère : la méthode se nomme **super**(liste des arguments) tout simplement.
2. faire les traitements spécifiques à la classe fille.

Si l'appel au constructeur de la classe mère n'est pas explicite, `Java` va essayer d'appeler automatiquement le constructeur par défaut (sans argument).

- S'il existe (ou si vous n'avez pas défini de constructeurs !), tout se passe bien
- Sinon, il faudra appeler le constructeur de la classe mère de façon explicite !

Exemple

```java
public class Personnage {
   private String nom;
   // Constructeur
   public Personnage(String name){
     this.nom = name;
   }

   public String presentation(){
     return "Je m'appelle" + nom;
   }
}
```

```java
public class Gaulois extends Personnage {

   public Gaulois(String name){
     super(name);
   }

   public String presentation(){
     return super.presentation() + " je suis un gaulois";
   }
}

public static void main(String[] args){
   Gaulois asterix = new Gaulois("Astérix");
   System.out.println( asterix.presentation());
}
```

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 partage 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 liasion 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 Romain extends Personnage {
2
3    public Romain(String name){ super(name);}
5    @Override public String presentation(){
6      return "romanus sum";
7    }

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

Tout objet hérite de la classe `Object` : conséquences

L'implémentation de toute méthode de la classe `Object` que vous ne redéfinissez pas sera évidemment l'implémentation de `Object` ! (doh !)

`toString()` : The `toString` method for class Object returns a string consisting of the name of the class of which the object is an instance, the at-sign character '@', and the unsigned hexadecimal representation of the hash code of the object. In other words, this method returns a string equal to the value of : `getClass().getName() + '@' + Integer.toHexString(hashCode())`

`protected clone()` :this method creates a new instance of the class of this object and initializes all its fields with exactly the contents of the corresponding fields of this object, as if by assignment ; the contents of the fields are not themselves cloned. Thus, this method performs a "shallow copy" of this object, not a "deep copy" operation.

Attention `clone()` est une méthode `protected` de la classe `Object`. Vous avez le droit de changer la visibilité pour votre classe (par exemple `public`).

Tout objet hérite de la classe `Object` : conséquences

`equals()` The equals method for class Object implements the most discriminating possible equivalence relation on objects ; that is, for any non-null reference values x and y, this method returns true if and only if x and y refer to the same object (x == y has the value true).

Il est donc de **votre responsabilité** de définir correctement la méthode `equals`.

attention : `boolean equals(Object obj)`
Le type de l'argument `obj` est `Object`.

Redéfinission en deux temps :

1. vérifiez si `obj` a le bon type[a]
2. si c'est le cas, on peut faire un transtypage (qui fonctionnera !), et on peut alors vérifier l'égalité des propriétés de l'objet

a. exactement le bon type ou bien une sous classe est-elle acceptable ?