

Introduction à Java

Cours 3: Programmation Orientée Objet en Java

Stéphane Airiau

Université Paris-Dauphine

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`.

Redéfinition 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 ?

Tout objet hérite de la classe `Object`

Modifier and Type	Method Description
<code>boolean</code>	<code>equals(Object obj)</code> Indicates whether some other object is "equal to" this one.
<code>Class<?></code>	<code>getClass()</code> Returns the runtime class of this <code>Object</code> .

discussion sur la méthode `equals` des classes `Box` et `TextBox`.

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**)

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.

Interfaces : définition

- Une interface s'écrit dans un fichier `.java` qui portera le même nom que l'interface.
- Le nom d'une interface commence toujours par une majuscule

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

- 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**.

Exemple

```
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 }
```

On peut maintenant utiliser l'interface comme un type normal !

```
Combattant soldat1 = new SoldatRomain();  
Combattant soldat2 = new SoldatRomain();  
Combattant obelix = new IrreductibleGaulois("Obélix");  
soldat1.attaque(obelix);  
soldat2.attaque(obelix);  
obelix.defend(soldat1);  
obelix.defent(soldat2);
```

Les objets sont déclarés avec l'interface `Combattant`.

On sait qu'on ne peut pas instancier l'interface !

Mais on peut créer des instances d'une classe qui implémente l'interface

➡ on sait qu'on peut faire appel aux méthodes de l'interface

➡ le code exécuté sera celui de la "véritable classe" de l'objet

un irréductible gaulois se bat surement différemment d'un romain.

Gestion des classes à l'aide de Packages

(espaces de noms)

Motivations

- garantir l'unicité des noms de classes.

ex : Par exemple, deux développeurs peuvent avoir l'idée de développer une classe `Dauphine`. Tant que les classes se trouvent dans des espaces de noms différents, cela ne posera aucun problème et les deux classes pourront être utilisables.

- nombre (parfois important) de classes dans un projet

➡ utiliser une structure hiérarchique de répertoires pour ordonner les classes

➡ on rassemble dans un package (~ répertoire) les classes reliées.

Déclaration d'une classe

- nom qualifié : nom de la classe à partir de la racine
- ➡ c'est le nom qualifié de la classe qui est unique
- pour les classes de la librairie standard, le nom qualifié commence par `java`, par exemple `java.util.Vector` est le nom qualifié de la classe `Vector`
- Attention, en Java, les espaces de noms ne s'emboîtent pas les uns dans les autres. `java.util` et `java.util.function` sont deux espaces de noms distincts avec chacun leurs classes et interfaces.

Dans le fichier source Java, on commence toujours à indiquer à quel espace de noms la classe appartient en commençant le fichier par la ligne **package** `<nom_du_package>`.

information redondante ➡ on connaît l'emplacement du fichier dans le disque.

Mais très utile lorsqu'on lit le code dans un éditeur !

Java vérifie la position du fichier dans le répertoire correspondant

Utiliser une classe d'un espace de noms

- Pour utiliser une classe qui se trouve dans le même espace de noms
 - utiliser son nom simple.
- Pour utiliser une classe à l'extérieur de son espace de noms
 - on peut toujours utiliser son nom qualifié.
mais cela peut être lourd
 - on fait un **import**
tout se passe comme si la/les classe(s) importée(s) font partie(s) de l'espace de nom courant.

NB L'import se fait toujours au début de la classe.

Attention

- 1- `import qqch.*`, vous importez seulement les classes, **pas les sous espaces** de noms.
 - ⇒ `import java.*` ne permet pas d'obtenir tous les espaces de noms qui commencent par `java..`
- 2- `import` de plusieurs espaces de noms
 - ⇒ car il est possible d'avoir des conflits de noms

L'utilisation de packages permet alors de désigner sans ambiguïté une classe.



On peut donc avoir deux méthodes avec exactement la même signature tant que ces deux méthodes appartiennent à des espaces de noms différents.

Librairie standard

- `java.lang` contient les classes fondamentales du langage.
- `java.util` contient les classes pour manipuler des collections d'objets, des modèles d'évènements, des outils pour manipuler les dates et le temps, et beaucoup de classes utiles.
- `java.io` contient les classes relatives aux entrées et sorties
- ...

Compilation et Exécution

Pour compiler une classe, il faut indiquer son chemin depuis la racine.

```
| javac important/premier/MaSecondeClasse.java
```

Le compilateur générera le fichier `important/premier/MaSecondeClasse`.
Si `MaSecondeClasse` possède une méthode `main`, on lance l'exécution en spécifiant le nom complet de la classe. Pour l'exemple, on lancerait donc

```
| java important.premier.MaSecondeClasse
```

Document le code

Cours 3: Programmation Orientée Objet en Java

Stéphane Airiau

Université Paris-Dauphine

Documenter le code

- javadoc est un outil qui va générer automatiquement de la documentation au format `html` à partir du code source
- La documentation du site <https://docs.oracle.com/javase/8/docs/api/> n'est rien d'autre que le résultat de javadoc sur le code source de la librairie standard de Java.

javadoc extrait de l'information sur

- les packages
 - les classes et les interfaces **public**
 - les variables **public** et **protected**
 - les méthodes et les constructeurs **public** et **protected**
- ➡ pour faire bien, on devrait décrire chacun de ces éléments dans un commentaire (qui sera utilisé par l'outil javadoc).

On place le commentaire juste avant l'élément qu'il décrit. Le commentaire commence par **/**** et se termine par ***/**.

Généralité

La première phrase sera le résumé (et sera vu comme tel par javadoc).

Comme en html ou xml, on va utiliser des marqueurs. Ceux-ci commenceront par le caractère @.

Comme on génère un code html, on peut utiliser des balises html à l'intérieur du commentaire. Par exemple `` pour l'emphase (italique), `` pour le texte en gras, `<code>` pour du code source, on peut même inclure des images, etc.

Evidemment, comme l'outil javadoc va gérer la mise en page, n'utilisez pas des balises pour les titres (`<h1>`, `<h2>`) ou pour `<hr>` pour placer une barre horizontale.

Commentaires pour les classes

- @author
- @version

```
1  /** A <code>Gaulois</code> objet is a Personnage who is a Gaulois
2      * @author Goscinny
3      * @author Uderzo
4      * @version 36.1
5      */
6  public class Gaulois extends Personnage {
7      ...
8  }
```

Commentaires pour les méthodes

- décrire chaque paramètre de la méthode avec un commentaire qui commence par la balise `@param`
- on décrit ce que retourne la méthode (quand elle n'est pas `void`) après la balise `@return`
- on décrit toute exception levée après la balise `@throws`

```
1  /** tells whether the gaulois thinks whether a fish is fresh
2     * @param fish the fish in question
3     * @return true when the gaulois thinks the fish is fresh,
4     * false otherwise
5     */
6  public boolean isFresh(Fish fish) {
7  }
```

Commentaires pour les variables

On n'a besoin de commenter seulement les variables **public**. Dans ce cas, il suffit d'utiliser un commentaire.

```
1 /** duration in time of the effect of the magic potion
2  */
3 public int magicPotionDuration;
```

- `@since` décrire la version à partir de laquelle l'élément disponible
- `@deprecated` indique que la classe, méthode ou variable ne devrait plus être utilisée
- `@see` cible va créer un lien vers la référence cible.
La cible peut être :

- une classe, une méthode ou une variable commentée

`@see`

`courseExamples.IrreductibleGaulois#fight(Fighter f)`
va faire un lien avec la méthode `fight(Fighter f)` qui se trouve dans la classe `IrreductibleGaulois` du package `courseExamples`.

- un lien html

`@see official
webpage`

Dans tout ce qu'on a vu jusqu'ici, il suffisait d'écrire des commentaires directement dans le code source.

Pour les packages, il faut écrire dans un fichier séparé dans chaque répertoire :

- un fichier `package-info.java` contient un commentaire javadoc décrivant le package
- on peut aussi fournir un fichier "overview" dans un fichier `overview.html`. Tout ce qui sera à l'intérieur des balises `<body>`
`</body>` sera utilisé par javadoc.

Générer la documentation

```
javadoc -d docDirectory package1 package2
```

La commande va générer la documentation pour les packages `package1`, `package2` et la placera dans le répertoire `-d docDirectory`.

Si on veut faire des liens automatiquement vers la document de la librairie standard de Java, on peut utiliser l'option `-link`

```
javadoc -link  
https://docs.oracle.com/javase/8/docs/api/ *.java
```