Introduction à Java

Cours 3: Programmation Orientée Objet en `Java`

Stéphane Airiau

Université Paris-Dauphine

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 ?

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.

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

```java
public interface Combattant {
   public void attaque(Personnage p);
   public void defend(Combattant c);
}
```

```java
public class IrreductibleGaulois implements Combattant {
     ...
   public void attaque(Personnage p){
     gourdePotionMagique.bois();
     while(p.isDebout())
       coupsDePoing(p);
   }

   public void defend(Combattant c){
     esquive();
     attaque(c);
   }
}
```

Utilisation

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

```
Combattant soldat1 = new SoldatRomain();
Combattant soldat2 = new SoldatRomain();
IrreductibleGaulois 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.

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`.

Documenter le code

`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 `<em>` pour l'emphase (italique), `<strong>` 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
4    */
5  public boolean isFresh(Fish fish){
6
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;
```

Autres commentaires

- `@since` décrire la version à partir de laquelle l'élémen 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 <a href="http://www.asterix.com">official webpage</a>`

Commentaire pour les packages

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
```