Complément de cours - La classe Object

Les méthodes de la classe `Object`

```
public  class Object {
     public Object() {...}  // contructeur

     public String toString() {...}

     protected  native Object clone()  throws CloneNotSupportedException
{...}

     public equals(java.lang.Object) {...}
     public native  int hashCode() {...}

     protected void finalize() throws Throwable {...}

     public  final  native Class getClass() {...}

     // méthodes utilisées dans la gestion des threads
     public  final  native  void notify() {...}
     public  final  native  void notifyAll() {...}

     public  final  void wait(long)  throws InterruptedException {...}
     public  final  void wait(long,  int)  throws InterruptedException
                                  {...}
}
```

Nous reviendrons en détails plus tard sur la signification des clauses `throws … Exception`. De même, les méthodes `notify()`, `notifyAll()`, `wait(long)` et `wait(long, int)` seront également revues en détails dans le chapitre sur les unités d'exécution (threads).

Une des possibilités offertes par l'héritage, est que tout objet instance d'une classe qui hérite d'une autre classe, peut utiliser les méthodes de cette autre classe. Donc, tout objet Java peut utiliser, sous certaines conditions, l'ensemble des méthodes de la classe `Object`. Certaines de ces méthodes sont très spécialisées, mais d'autres sont utilisées très fréquemment. D'autres enfin, sont utilisées de manière spéciale par la machine Java, et ont donc un statut particulier.

2.1. La méthode `toString()`

La première de ces méthodes est la méthode `toString()`. Cette méthode est utilisée par la machine Java toutes les fois où elle a besoin de représenter un objet sous forme d'une chaîne de caractères. Par exemple, il est légal d'écrire la ligne suivante :

```
Marin marin =  new Marin ("Surcouf",  "Robert",  25000) ;

System.out.println("Marin : " + marin) ;
```

La méthode `println` de l'objet `System.out` prend en paramètre un objet de type `String`. La machine Java a donc besoin de convertir `("Marin : " + marin)` en objet de type `String`. Cela passe par la conversion de l'objet `marin` en objet `String`. Cette conversion s'effectue par appel à la méthode `toString()` de la classe `Object`.

Il est donc presque équivalent d'écrire le code précédent et celui-ci :

```
System.out.println("Marin : " + marin.toString()) ;
```

En apparence, les deux codes sont équivalents. En fait, en toute rigueur, le second échouera avec une ignoble NPE si l'objet marin est nul, alors que le premier affichera `null`.

Dans la pratique, on peut se demander ce que ce code va nous afficher. La réponse est immédiate si l'on tente de l'exécuter : une chaîne de caractères qui va ressembler à `Marin@b82e3f203`. Les chaînes de caractères retournées par la méthode `toString()` de la classe `Object` ont toutes cette forme : le nom de la classe, suvie du caractère `@`, et une adresse en hexadécimal, qui est l'adresse mémoire où l'objet considéré est enregistré. Cela garantit l'unicité de ce code en fonction de l'objet, ce qui a son importance. En tout cas, il apparaît clairement que ce résultat c'est pas très sympathique, et en tout cas peu explicite pour un utilisateur humain normalement constitué.

Heureusement pour nous, il est possible de changer ce comportement, et d'afficher une chaîne de caractères plus explicite. On utilise pour cela un mécanisme qui s'appelle la "surcharge". Sans entrer dans les détails de ce mécanisme, disons pour le moment qu'il est possible de réécrire cette méthode `toString()` dans la classe `Marin`. Il suffit pour cela d'ajouter les lignes de code suivantes à la classe que nous avons déjà écrite.

Exemple 4. Surcharge de `toString()` dans la classe `Marin`

```
public String toString() {
    String resultat =  super.toString() ;
    resultat +=  "\nNom : " + nom ;
    resultat +=  "\nPrénom : " + prenom ;
    resultat +=  "\nSalaire : " + salaire ;
     return resultat ;
}
```

Cette méthode commence par un appel à la méthode `super.toString()`. Cette syntaxe signifie qu'elle appelle une méthode `toString()` qui doit être définie parmi les super-classes de `Marin`. Sur notre exemple simple, cette classe n'en étend aucune autre par déclaration, elle étend donc `Object`. Il se trouve par chance que la classe `Object` possède une méthode `toString()`, c'est donc celle-là qui va être appelée tout d'abord. Si cela n'avait pas été le cas, on aurait eu une erreur de compilation.

Les lignes suivantes ajoutent des éléments supplémentaires. Le résultat de l'appel à cette méthode `toString()` devra ressembler à ceci :

```
Marin@b82e3f203
Nom : Surcouf
Prénom : Robert
Salaire : 25000
```

2.2. La méthode `equals()`

Cette méthode permet, comme son nom l'indique, de comparer deux objets, et notamment de savoir s'ils sont égaux. Un peu plus haut nous avons créé deux objets de la classe `Marin`, qui possédaient même nom, même prénom et même salaire, et nous les avons comparés avec `==`. Nous avions alors vu que `==` comparait les adresses mémoire des objets, et dans ce cas renvoyait `false`. Ce comportement est logique, mais il serait utile d'avoir à disposition un moyen de comparer des objets qui puisse nous dire que si leurs champs sont égaux, alors ces objets sont égaux. En d'autres

termes, remplacer une égalité technique en égalité sémantique. C'est l'objet de la méthode `equals()`.

Écrivons une méthode `equals()` pour notre classe `Marin`, qui retourne `true` si les champs des deux objets comparés sont égaux.

Exemple 6. Surcharge de `equals()` dans la classe `Marin`

```
public  boolean equals(Object o) {
     if (!(o  instanceof Marin))  // retourne false si l'objet o est nul
         return false ;

    Marin marin = (Marin)o ;  //forcer l'objet o à être de type Marin

     return nom.equals(marin.nom) &&
           prenom.equals(marin.prenom) &&
           salaire == marin.salaire ; }
```

Remarquons tout d'abord que l'opérateur instanceof retourne ystématiquement false si l'objet testé est nul. Cela garantit que l'objet marin est non nul.

Tout d'abord, remarquons que la méthode `equals()` prend en paramètre un objet de type `Object`. Une erreur commune consisterait à déclarer l'objet passé en paramètre comme devant être de type `Marin`. Cette erreur est un peu subtile, et nous la détaillerons dans la suite. Disons ici qu'il est absolument nécessaire que cette méthode `equals()` prenne un `Object` en paramètre.

La première chose à faire est de tester si l'objet passé en paramètre est non nul, et s'il est bien de la classe `Marin`. La comparaison d'un objet de type `Marin` avec un objet nul ou d'un autre type est légale, et elle retournera `false` systématiquement, ce qui est normal.

L'opérateur `instanceof`, utilisé ligne 5, permet de tester la classe d'un objet. En l'occurrence, il retourne `true` pour tous les objets de la classe `Marin`, et de toute classe qui hériterait de `Marin`.

Une fois que nous sommes sûr d'avoir un objet `Marin` en paramètre, alors il nous faut comparer ses champs un par un. Pour pouvoir accéder à ses champs, il faut le convertir en objet de la bonne classe, c'est ce que fait la ligne 8. Cette opération s'appelle un *cast* , elle consiste à déclarer un objet (ici `marin`), et à lui affecter la valeur de l'objet à convertir, en mettant devant et entre parenthèses le type dans lequel on veut faire cette conversion. Il faut toujours vérifier le type de l'objet que l'on caste à l'aide d'un `instanceof`, avant de faire le *cast* (il existe également une autre méthode, que nous verrons dans la suite). Un *cast* réalisé sur un objet qui ne serait pas de la bonne classe jetterait une exception.

La comparaison des champs de `marin` est faite entre les lignes 10 et 12. On remarquera que la comparaison des chaînes de caractères se fait en utilisant aussi la méthode `equals()`, de la classe `String`. La classe `String` possède sa propre méthode `equals()`, qui retourne `true` si les deux chaînes de caractères contiennent les mêmes caractères.

En toute rigueur, avant de tester l'égalité de `nom` et `prenom`, il nous faudrait tester si ces deux chaînes de caractères sont nulles ou pas. Si `nom` est nul (par exemple), alors notre méthode `equals()` échouera, en jetant l'ignoble NPE. Comme on peut le voir, l'écriture d'une méthode `equals()` correcte et complète est un processus qui mérite de l'attention.

2.3. La méthode `hashCode()`

Le rôle de la méthode `hashCode()` est de calculer un code numérique pour l'objet dans lequel on se trouve. Ce code numérique est censé être représentatif de l'objet, nous allons expliciter ce point immédiatement.

Techniquement, la méthode `hashCode()` est une méthode native (une méthode `native` est une méthode qui n'est pas écrite en Java, mais dans un autre langage, qui peut être le C, le C++, ou tout autre langage) qui permet de calculer un nombre (`int`) unique associé à une instance de n'importe quelle classe. Par défaut, la méthode de la classe `Object` retourne l'adresse à laquelle est rangé cet objet, nombre effectivement unique, puisqu'on ne peut pas ranger deux objets au même endroit en mémoire. En toute rigueur, ce point dépend de la JVM que l'on utilise, mais c'est le cas dans la JVM de Sun.

Le point délicat est le contrat qui lie les méthodes `equals()` et `hashCode()` dans les spécifications de Java.

- Deux objets égaux au sens de `equals()` doivent retourner le même `hashCode()` ;
- Il n'est pas nécessaire que deux objets différents au sens de `equals()` retournent deux `hashCode()` différents...

Donc, surcharger la méthode `equals()` d'une classe entraîne systématiquement la surcharge de la méthode `hashCode()`. Ne pas respecter cette règle revient à s'exposer à des bugs obscurs et très difficiles à corriger, nous verrons des exemples précis dans la suite.

Surcharger une méthode `hashCode()` se fait en respectant un algorithme précis. Il existe plusieurs variantes de cet algorithme, nous en donnons une ici. Supposons que nous ayons surchargé une méthode `equals()`, en écrivant l'égalité entre plusieurs champs de notre classe.

La première chose à faire est de choisir deux nombres entiers, pas trop petits, 17 et 31 peuvent faire l'affaire. On initialise l'algorithme en prenant `hashCode = 17`.

Pour chacun des autres champs `c` pris en compte par la méthode `equals()`, on construit l'entier `hash` suivant :

- si `c` est un booléen, `hash` vaut 1 si `c` est `true`, 0 s'il est `false` ;
- si `c` est de type `byte`, `short`, `int` ou `char`, alors `hash` vaut `(int)c` ;
- si `c` est de type `long`, alors `hash` vaut `(int)(c^(c >>> 32))` ;
- si `c` est de type `float`, alors `hash` vaut `Float.floatToIntBits(f)` ;
- si `c` est de type `double`, alors `hash` vaut `Double.doubleToLongBits(f)`, et l'on prend le code de hachage du `long` que l'on récupère ;
- si `c` est nul alors `hash` vaut 0 ;
- si `c` est un `objet` non nul, alors `hash` vaut `c.hashCode()` ;
- si `c` est un tableau, alors chacun des éléments du tableau est traité comme un champ à part entière.

Et pour chacun de ces champs, on met à jour `hashCode` :

```
hashCode =  31 * hashCode + hash
```

Pour notre classe `Marin`, la méthode `hashCode()` est la suivante.

Exemple 7. Surcharge de `hashCode()` dans la classe `Marin`

```
public  int hashCode() {
     int hashCode =  17 ;
    hashCode =  31 * hashCode + ((nom == null) ?  0 : nom.hashCode()) ;
    hashCode =  31 * hashCode + ((prenom == null) ?  0 : prenom.hashCode())
;
    hashCode =  31 * hashCode + salaire ;
     return hashCode ; }
```

Notons que cette implémentation de la méthode hashCode() est une implémentation parmi d'autres. En particulier, les nombres entiers choisis peuvent varier.

2.4. La méthode `getClass()`

Cette méthode retourne un objet, instance d'une classe particulière appelée `Class`. Tout est objet en Java, y compris les classes elles-mêmes ! Il existe donc une classe `Class`, qui modélise les classes Java. Nous verrons par la suite que cette classe est à la base des mécanismes d'introspection, et ouvre la porte à des méthodes de programmation très puissantes.

Notons que la méthode `toString()` de la classe `Class` est surchargée, mais ne retourne pas le nom de la classe, comme on pourrait s'y attendre.

```
Marin m =  new Marin("Surcouf",  "Robert") ;
System.out.println("Classe de marin : " + m.getClass()) ;
> Classe de marin : class Marin
```

Si l'on veut juste le nom de la classe, il faut invoquer sa méthode `getName()`.

```
Marin m =  new Marin("Surcouf",  "Robert") ;
System.out.println("Classe de marin : " + m.getName()) ;
> Classe de marin : Marin
```

2.5. La méthode `finalize()`

La présentation de la méthode `finalize()` est une bonne occasion de parler de la destruction des objets. Effectivement, nous avons vu comment construire des objets, mais rien n'a été dit sur leur destruction. Et pour cause : en Java, il n'y a rien à faire pour détruire un objet, la notion de destructeur n'existe tout simplement pas. La machine Java fonctionne avec un ramasse-miettes (*garbage collector* en anglais), qui est censé détecter les objets qui ne sont plus référencés par aucune variable, et les effacer lui-même. Idéalement, lorsque la dernière référence vers un objet est coupée, le ramasse-miettes enregistre cet objet, et de temps en temps, il se met en marche et libère la mémoire.

Dans la réalité, les choses sont en fait très délicates, et si l'on n'y prend pas garde, le ramasse-miettes oublie des objets, créant ainsi des fuites de mémoire.

C'est à cet instant qu'il invoque la méthode `finalize()`. Cette méthode est donc un callback, appelé par la machine Java juste avant l'effacement d'un objet. Notons tout de suite que l'appel de cette méthode ne se fait pas au moment où un objet n'est plus référencé, mais au moment où la machine Java décide de l'effacer. On n'a donc aucune maîtrise sur le temps au bout duquel l'objet est effacé, et sur le temps au bout duquel `finalize()` est appelé.

Le fonctionnement de la libération de la mémoire, expliqué de cette façon, est assez idéal : normalement, aucune fuite de mémoire ne peut avoir lieu... Sauf que ce mécanisme ne fonctionne

pas dans tous les cas. La gestion de la libération de la mémoire en Java est un mécanisme complexe, qui a subi de nombreuses évolutions et optimisations au fil des versions, mais qui n'est toujours pas parfait. Il existe des cas dans lesquels un paquet d'objets qui se référencent entre eux, ne sont plus référencés par rien, et ne sont pas effacés par la machine Java. C'est là que la méthode `finalize()` peut être utile, en coupant des relations entre objets, de façon à aider la machine Java à détecter les objets à supprimer.

Donc : les fuites de mémoire existent en Java, même si le mécanisme d'allocation et de libération de mémoire mis au point dans les dernières versions de JVM se révèlent très performant, en fait plus performant même que la classique allocation / libération manuelle du C ou du C++.

2.6. La méthode `clone()`

La méthode `clone()` est une méthode déclarée `native`. Une méthode `native` est une méthode qui n'est pas écrite en Java, mais dans un autre langage, qui peut être le C, le C++, ou tout autre langage. Java utilise un mécanisme spécial pour éventuellement passer des paramètres aux méthodes natives, les invoquer, et récupérer ce qu'elles retournent. Une méthode native n'est en général pas portable d'une machine à l'autre, on perd donc un des intérêts majeurs de Java en écrivant des méthodes natives. Ici, cette méthode fait partie de l'API standard, qui de toute façon existe pour toutes les machines / OS existantes.

Le rôle de la méthode `clone()` est de dupliquer un objet rapidement, en dupliquant la zone mémoire dans laquelle il se trouve à l'aide d'un processus rapide. Pour cloner un objet, il suffit donc d'appeler cette méthode, qui nous renverra une copie conforme de cet objet.

Attention toutefois le clonage des objets est interdit par défaut. Afin de l'utiliser, il faut surcharger la méthode `clone()` de la classe `Object`, qui est `protected`, par une méthode `public` de la classe de l'objet que l'on veut cloner. En plus, il faut que la classe dont on veut cloner les instances, implémente l'interface `Cloneable`. Cette interface ne comporte pas de méthode, elle est juste là pour autoriser le clonage. Tenter de cloner un objet qui n'implémente pas cette interface génèrera une exception de type `CloneNotSupportedException` (moins ignoble que l'ignoble NPE, mais tout de même...).

La surcharge de cette méthode n'a pour objet que d'exposer publiquement la méthode `clone()` de la classe `Object`. Rendre une classe `Marin` clonable, peut donc se faire de la façon suivante.

Exemple 5. Surcharge de `clone()` dans la classe `Marin`

```
public  class Marin
 implements Cloneable {  // déclaration indispensable
    // ici on propage l'exception, on aurait pu aussi
    // l'attraper localement
    public Object clone()  throws CloneNotSupportedException {
       return  super.clone() ;
   } }
```

Notons que l'on peut surcharger la méthode `clone()` par une méthode qui ne jette aucune exception, dans la mesure où la clause `throws` ne fait pas partie de la signature d'une méthode. Dans ce cas, l'exception que peut jeter l'appel à `super.clone()` doit être attrapée localement.