UNIVERSITÉ DE REIMS CHAMPAGNE-ARDENNE

Licence 1 INFO
INFO0201

Cours à distance

La classe `Object` et ses méthodes

1 La classe `Object`

En *Java*, toute classe qui n'hérite pas d'une autre classe en utilisant le mot clé `extends` hérite implicitement de la classe `Object` qui se trouve dans le package `java.lang` (qui lui aussi est toujours importé implicitement). Ainsi *Java* fonctionne comme une hiérarchie et toutes les classes *Java* ont comme premier ancêtre la classe `Object`.

Etant donné que toutes les classes héritent (directement ou non) de la classe `Object`, elles héritent donc des membres (attributs et méthodes) de la classe `Object`, et donc tout objet a accès aux méthodes publiques définies dans la classe `Object`. Certaines des méthodes de cette classe sont très intéressantes et utiles. Vous avez déjà rencontré quelques unes de ces méthodes comme `toString` qu'on vous demande d'ajouter à chaque classe, jusqu'à présent vous utilisiez la signature imposée sans trop comprendre, maintenant vous comprenez qu'il s'agit d'une surcharge de la méthode se trouvant dans la classe `Object`. Cette classe se trouvant tout en haut de la hiérarchie, elle possède donc des méthodes très générales comme nous allons le voir.

Comme toutes les classes héritent (directement ou non) de la classe `Object`, lorsqu'on définit le constructeur d'une classe si l'appel au constructeur de la super-classe n'est pas clairement explicité, c'est le constructeur par défaut qui est appelé implicitement : `super()`. La class `Object` possède donc un constructeur par défaut.

2 La méthode `toString`

Signature : `public String toString()`

La méthode `toString` est une méthode très utile, elle permet d'obtenir une représentation sous forme de chaîne de caractères d'un objet. Elle est implicitement appelée par le compilateur lorsqu'on concatène une chaîne de caractères avec un objet ou lorsqu'on passe un objet dans une méthode attendant une chaîne de caractères comme par exemple dans la méthode `System.out.println(...)`.

Par défaut l'implémentation de la méthode `toString` dans la classe `Object` retourne le type de l'objet suivi de @ suivi du code de hachage de l'objet, par exemple *Point@c05fbe517a*. Il suffit de redéfinir cette méthode pour obtenir la représentation souhaitée.

Exemple :

```java
public class Point{
    ...
    public String toString (){
        return super.toString() +
            "(" + this.x + ";" +
            this.y + ")";
    }
}
```

```java
public class TestPoint{
    Point p = new Point(2.5,3);
    System.out.println(p);
}
```

Affichage : `Point@c05fbe517a(2.5;3)`

3 La méthode `clone`

Signature : `protected native Object clone()`

La méthode `clone` est utilisée pour cloner une instance, c'est-à-dire obtenir une copie d'un objet, c'est une copie de bas niveau efficace. Par défaut, elle est déclarée `protected` car toutes les classes ne désirent pas permettre de cloner une instance. Pour permettre le clonage d'une classe il faut donc la redéfinir dans la classe et la rendre `public`.

Pour qu'un objet soit clonable, sa classe doit implémenter l'interface marqueur `Cloneable` sinon on a une exception `CloneNotSupportedException`. L'interface `Cloneable` ne possède pas de méthode, elle autorise seulement le clonage.

L'implémentation par défaut de la méthode `clone` ne réalise pas un clonage en profondeur, elle n'appelle pas les constructeurs pour créer la nouvelle instance. Cela signifie que si les attributs de la classe d'origine référencent des objets, les attributs du clone référenceront les mêmes objets. Si ce comportement n'est pas celui désiré, alors il faut fournir une nouvelle implémentation de la méthode clone dans la classe.

Exemple pour permettre à une classe d'être clonée en utilisant l'implémentation par défaut, c'est-à-dire une copie de surface :

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

Exemple pour permettre à une classe d'être clonée en réalisant une copie profonde, il faut cloner les composants, pour cela on peut utiliser la méthode `clone` de chacun des objets ou bien le constructeur par copie si le composant n'est pas clonable :

```
class Personne implements Cloneable {
    ...
    public Object clone() {
        Personne ref = null ;
        try {
            ref = (Personne) super.clone();
            ref.dateNaiss = ((Date) dateNaiss).clone();
        }catch ( CloneNotSupportedException e ) {
            e.printStackTrace(System.err);
        }
        return ref ;
    }
}
```

4 La méthode `equals`

Signature : `public boolean equals (Object ref)`

Comme vous le savez, en `Java`, l'opérateur == sert à comparer les références. Il ne faut donc jamais l'utiliser pour comparer des objets. La comparaison d'objets se fait grâce à la méthode `egalA` que nous avons déjà vu ou par la méthode `equals` héritée de la classe `Object`.

L'implémentation par défaut de la méthode `equals` fournie par la classe `Object` compare les références entre elles, il faut donc la redéfinir pour comparer "vraiment" l'objet passé en paramètre à l'objet courant. La signature de la méthode `equals` impose que le paramètre soit de type `Object`, il est donc possible de comparer tout type d'objet entre eux puisque tous héritent d'`Object` et donc tous peuvent être référencés dans une variable de type `Object` avec le typage dynamique. Il est donc important de commencer par vérifier que le paramètre est d'un type acceptable pour la comparaison, pour cela il faut se servir de l'introspection et donc de `instanceof`. Une fois le type vérifié, on utilise le transtypage pour accéder aux membres, il n'y a plus qu'à comparer les membres en fonction de ce que l'on souhaite accepter comme égalité. Si les membres sont des objets, il faut là aussi les comparer avec `equals`.

Exemple avec des attributs de type primitif :

```
public class Point { ...
   public boolean equals(Object obj) {
      boolean res = false;
      if (obj instanceof Point){
         Point ref = (Point)obj ;
         res = (x==ref.x && y==ref.y);
      }
      return res;
   }
}
```

Exemple avec des attributs de type objet (non-primitif) :

```
public class Vehicule {
   private String immatriculation;
   private Marque marque;
   public Vehicule(String immatriculation, Marque marque) {
      this.immatriculation = immatriculation;
      this.marque = new Marque(marque);
   }
   public boolean equals(Object obj) {
      boolean res = false;
      if (! (obj instanceof Vehicule)) { res = false; }
      else{
         Vehicule v = (Vehicule) obj;
         res = this.marque.equals(v.marque) &&
            this.immatriculation.equals(v.immatriculation);
      }
      return res;
  } ...
}
```

Que se passe-t-il si la référence passée en paramètre vaut *null* ? `instanceof` retourne alors *false* donc la référence n'est pas considérée comme étant un type comparable à l'objet courant.

Attention, l'implémentation de `equals` doit être conforme à certaines règles pour s'assurer qu'elle fonctionnera correctement :

— être réflexive :

C'est-à-dire que pour un objet `x` non nul : `x.equals(x)` doit être vrai.

— être transitive :

C'est-à-dire que pour des objets `x`, `y` et `z` non nuls :
Si `x.equals(y)` est vrai ET `y.equals(z)` est vrai
Alors `x.equals(z)` doit être vrai

— être symétrique :

Si `x.equals(y)` est vrai
Alors `y.equals(x)` doit être vrai

Une attention particulière devra être donnée à la symétrie en particulier dans le cas de comparaison entre classes mère et fille. En effet `instanceof` renvoie vrai lorsque l'instance courante est une fille de la classe testée, par exemple si `Moto` est une fille de `Vehicule`, considérant `m` une instance de `Moto`, alors `m instanceof Vehicule` retourne vrai, mais la réciproque est bien sûr fausse. Il est donc possible de se retrouver avec un `Vehicule` égal à une `Moto` alors que si l'on teste si une `Moto` est egal à un `Vehicule` la réponse sera non. Pour palier ce problème, on utilise la méthode `getClass()` expliquée ci-après afin de vérifier que les objets sont bien du même type.

5 La méthode `hashCode`

Signature : `public native int hashCode()`

La méthode `hashCode` renvoie un code numérique unique pour l'objet, elle est fournie pour l'utilisation de certains algorithmes, notamment pour l'utilisation des tables de hachage. Le principe d'un algorithme de hachage est d'associer un identifiant à un objet. Cet identifiant doit être le même pour la durée de vie de l'objet. De plus, deux objets égaux doivent avoir le même code de hachage.

L'implémentation de cette méthode peut se révéler assez technique. En général, on se basera sur les attributs utilisés dans l'implémentation de la méthode `equals` pour en déduire le code de hachage.

Cette méthode ne doit être redéfinie que si cela est réellement utile. Par exemple si une instance de cette classe doit servir de clé pour une instance de `HashMap`.

6 La méthode `finalize`

Signature : `protected void finalize() throws Throwable`

La méthode `finalize` est appelée par le ramasse-miettes (*garbage collector* avant que l'objet ne soit supprimé et la mémoire récupérée. Redéfinir cette méthode donne donc l'opportunité au développeur de déclencher un traitement avant que l'objet ne disparaisse. Cependant le ramasse-miettes ne s'effectue pas obligatoirement dès que la référence est supprimée donc ne donne aucune garantie sur le moment où cette méthode sera appelée.

Cette méthode ne doit pas être appelée par nous même. Normalement elle permet de tout arrêter "proprement', dans la pratique, elle est dépréciée depuis *Java9*.

7 La méthode `getClass`

Signature : `public final native Class getClass()`

La méthode `getClass` retourne un objet, instance d'une classe particulière appelée `Class` qui représente la classe de l'instance. Cela signifie qu'un programme *Java* peut accéder à la définition de la classe d'une instance. Cette méthode est à la base des mécanismes d'introspection et est notamment très utilisée dans des usages avancés impliquant la réflexivité.

L'objet de type `Class` obtenu permet d'accéder à différentes informations (voir l'API pour les connaître toute !) par exemple :

```
Point p = new Point(-3.1, 2.5);
System.out.println("getClass␣de␣Point␣:␣" + p.getClass());
System.out.println("getName␣de␣Point␣:␣" + p.getClass().getName());
```

Affichage :

```
getClass de Point : class Point
getName de Point : Point
```