La classe Object

Java est un langage qui ne supporte que l'héritage simple. L'arborescence d'héritage est un arbre dont la racine est la classe Object. Si le développeur ne précise pas de classe parente dans la déclaration d'une classe, alors la classe hérite implicitement de Object.

La classe Object fournit des méthodes communes à toutes les classes. Certaines de ces méthodes doivent être redéfinies dans les classes filles pour fonctionner correctement.

> Rien ne vous interdit de créer une instance de Object.

```
Object myObj = new Object();
```

La méthode equals

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 equals héritée de la classe Object.

```
Vehicule v1 = new Voiture("DeLorean");
Vehicule v2 = new Moto("Kaneda");

if (v1.equals(v1)) {
  System.out.println("v1 est identique à lui-même.");
}

if (v1.equals(v2)) {
  System.out.println("v1 est identique à v2.");
}
```

L'implémentation par défaut de equals fournie par Object compare les références entre elles. L'implémentation par défaut est donc simplement :

```
public boolean equals(Object obj) {
  return (this == obj);
```

```
}
```

> Il ne faut pas déduire de l'implémentation par défaut qu'il est possible d'utiliser == pour comparer des objets. N'importe quelle classe héritant de la classe Object peut modifier ce comportement : à commencer par une des classes les plus utilisée en Java, la classe String.

Parfois, l'implémentation par défaut peut suffire. C'est notamment le cas lorsque l'unicité en mémoire suffit à identifier un objet. Cependant, si nous ajoutons la notion de plaque d'immatriculation à notre classe *Vehicule* :

```
public class Vehicule {

  private String immatriculation;
  private final String marque;

  public Vehicule(String immatriculation, String marque) {
    this.immatriculation = immatriculation;
    this.marque = marque;
  }

  // ...

}
```

Alors l'attribut immatriculation introduit l'idée d'identification du véhicule. Il est donc judicieux de considérer que deux véhicules sont égaux s'ils ont la même immatriculation. Dans ce cas, il faut redéfinir la méthode equals.

```
package com.cgi.udev.conduite;

public class Vehicule {

  private String immatriculation;
  private final String marque;

  public Vehicule(String immatriculation, String marque) {
    this.immatriculation = immatriculation;
    this.marque = marque;
  }
```

```
  @Override
  public boolean equals(Object obj) {
    if (! (obj instanceof Vehicule)) {
      return false;
    }
    Vehicule vehicule = (Vehicule) obj;
    return this.immatriculation != null &&
           this.immatriculation.equals(vehicule.immatriculation);

  }

  // ...

}
```

Dans l'exemple précédent, notez l'utilisation de **instanceof** pour vérifier que l'objet en paramètre est bien compatible avec le type *Vehicule* (sinon la méthode retourne **false**). En effet, la signature de equals impose que le paramètre soit de type Object. Il est donc important de commencer par vérifier que le paramètre est d'un type acceptable pour la comparaison. Notez également, que l'implémentation est telle que deux véhicules n'ayant pas de plaque d'immatriculation ne sont pas identiques.

L'implémentation de equals doit être conforme à certaines règles pour s'assurer qu'elle fonctionnera correctement, notamment lorsqu'elle est utilisée par l'API standard ou par des bibliothèques tierces.

- Son implémentation doit être réflexive :
  Pour x non nul, x.equals(x) doit être vrai
- Son implémentation doit être symétrique :
  Si x.equals(y) est vrai alors y.equals(x) doit être vrai
- Son implémentation doit être transitive :
  Pour x, y et z non nuls
  Si x.equals(y) est vrai
  Et si y.equals(z) est vrai
  Alors x.equals(z) doit être vrai
- Son implémentation doit être consistante
  Pour x et y non nuls
  Si x.equals(y) est vrai alors il doit rester vrai tant que l'état de x et de y est inchangé.
- Si x est non nul alors x.equals(null) doit être faux.

> Il est parfois facile d'introduire un bug en Java.

```
if (x.equals(y)) {
  // ...
}
```

Le code ci-dessus ne teste pas la possibilité pour la variable *x* de valoir **null**, entraînant ainsi une erreur de type NullPointerException. Il ne faut donc pas oublier de tester la valeur **null** :

```
if (x != null && x.equals(y)) {
  // ...
}
```

Lorsque l'un des deux termes est une constante, alors il est plus simple de placer la constante à gauche de l'expression de façon à éviter le problème de la nullité. En effet, equals doit retourner **false** si le paramètre vaut **null**. Cela est notamment très pratique pour comparer une chaîne de caractères avec une constante :

```
if ("Message à comparer".equals(msg)) {
  // ...
}
```

On peut aussi utiliser la classe outil java.util.Objects qui fournit la méthode de classe equals(Object, Object) pour prendre en charge le cas de la valeur **null**. Notez toutefois que equals(Object, Object) retourne **true** si les deux paramètres valent **null**.

La méthode hashCode

La méthode hashCode est fournie pour l'utilisation de certains algorithmes, notamment pour l'utilisation de table 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.

```
package com.cgi.udev.conduite;

public class Vehicule {

  private String immatriculation;
  private final String marque;

  public Vehicule(String immatriculation, String marque) {
    this.immatriculation = immatriculation;
    this.marque = marque;
  }

  @Override
  public boolean equals(Object obj) {
    if (! (obj instanceof Vehicule)) {
      return false;
    }
    Vehicule vehicule = (Vehicule) obj;
    return this.immatriculation != null &&
           this.immatriculation.equals(vehicule.immatriculation);

  }

  @Override
  public int hashCode() {
    return immatriculation == null ? 0 : immatriculation.hashCode();
  }

  // ...

}
```

La méthode toString

La méthode toString est une méthode très utile, notamment pour le débugage et la production de log. 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.

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. Il suffit de redéfinir cette méthode pour obtenir la représentation souhaitée.

```
package com.cgi.udev.conduite;

public class Vehicule {

  private final String marque;

  public Vehicule(String marque) {
    this.marque = marque;
  }

  @Override
  public String toString() {
    return "Véhicule de marque " + marque;
  }

  // ...

}
```

```
Vehicule v = new Vehicule("DeLorean");

String msg = "Objet créé : " + v;

System.out.println(msg); // "Objet créé : Véhicule de marque DeLorean"
```

La méthode finalize

La méthode finalize est appelée par le ramasse-miettes 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, nous avons déjà vu dans le chapitre sur le cycle de vie que le fonctionnement du ramasse-miettes ne donne aucune garantie sur le fait que cette méthode sera appelée.

La méthode clone

La méthode clone est utilisée pour cloner une instance, c'est-à-dire obtenir une copie d'un objet. Par défaut, elle est déclarée **protected** car toutes les classes ne désirent pas permettre de cloner une instance.

Pour qu'un objet soit clonable, sa classe doit implémenter l'interface marqueur Cloneable. L'implémentation par défaut de la méthode dans Object consiste à jeter une exception CloneNotSupportedException si l'interface Cloneable n'est pas implémentée. Si l'interface est implémentée, alors la méthode crée une nouvelle instance de la classe et affecte la même valeur que l'instance d'origine aux attributs de la nouvelle instance. L'implémentation par défaut de clone n'appelle pas les constructeurs pour créer la nouvelle instance.

> L'implémentation par défaut de la méthode clone ne réalise pas un clonage en profondeur. 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.

> Par défaut, tous les tableaux implémentent l'interface Cloneable et redéfinissent la méthode clone afin de la rendre **public**. On peut donc directement cloner des tableaux en Java si on désire en obtenir une copie.

```
int[] tableau = {1, 2, 3, 4};
int[] tableauClone = tableau.clone();
```

La méthode getClass

La méthode getClass permet d'accéder à l'objet représentant la classe de l'instance. Cela signifie qu'un programme Java peut accéder par programmation à la définition de la classe d'une instance. Cette méthode est notamment très utilisée dans des usages avancés impliquant la *réflexivité*.

L'exemple ci-dessous, affiche le nom complet (c'est-à-dire en incluant son package) de la classe d'un objet :

```
Vehicule v = new Vehicule("DeLorean");


System.out.println(v.getClass().getName());
```

Les méthodes de concurrence

La classe Object fournit un ensemble de méthodes qui sont utilisées pour l’échange de signaux dans la programmation concurrente. Il s’agit des méthodes notify, notifyAll et wait.

Revision #1
Created 10 February 2025 12:13:25 by Nicolas
Updated 10 February 2025 12:14:52 by Nicolas