Null, equals et toString

Dans ce chapitre on va parler des références (addresses en mémoire) ainsi que du fonctionnement du `.equals()` et du `.toString()`

Qu'est ce qu'une référence "null"

Plus tot, en particulier avec les tableaux, on pouvait se retrouver avec une valeur `null` symolisant l'absence de valeur.

En réalité une valeur null symbolise plus précisément l'absence d'addresse (l'absence de référence). Et il est très important de toujours prendre en compte les cas dans le programme où une valeur nulle pourrait être retournée et la traiter en fonction.

```
// Ici on crée un tableau de Strings d'une longueur de 5 vide
String[] monTableau = new String[5];

// Ici on demande de retourner l'élement en position 0 sans l'avoir redéfini
System.out.println(monTableau[0]);
// On se retrouve avec "null" car il n'y a aucune valeur dans cette position du tableau
```

== vs .equals()

Comme vu dans le chapitre sur les Strings le `.equals()` n'est pas la même chose que l'opérateur booléen `==` sur les objets.

La raison est que `==` va comparer les références tandis que `.equals()` va comparer les valeurs des propriétés.

```
// Si deux chaines ont la même valeur lors de leur définitons, ils auront la même addresse en
mémoire
String helloA = "hello world";
String helloB = helloA;
String helloC = "hello world";

if (helloA == helloB && helloB == helloC) {
```

```
    System.out.println("Les références de A, B et C sont les mêmes.");
}

// En revanche si on fait des transformations sur ce String (même si il a au final la même valeur), alors l'addresse sera différente
String helloD = helloA.toUpperCase().toLowerCase();

if (helloA != helloD && helloA.equals(helloD)) {
    System.out.println("Les références de A et D ne sont pas les mêmes.");
    System.out.println("En revanche les deux ont la même valeur.");
}

// Ainsi le == compare les références
// Tandis que le .equals compare les valeurs
```

Héritage et Object

Quand on crée une classe en Java, notre classe va automatiquement "hériter" tout un tas de méthodes et/ou propriétés d'une autre clase par défault dans Java appellée "Object".

Nous allons donc avoir certaines méthodes par défault tel que .equals ou .toString. Mais il faut en général redéfinir ces derniers.

Redéfinition du .equals()

La raison pour laquelle il faut redéfinir le .equals est que par défault, il va avoir le même effet que `==` (comparer les références plus tot que les valeurs). Donc voici un exemple de redéfinition :

```
// @Override indique que l'on va redéfinir une méthode (celle de Object en l'occurence)
// Java va tester au moment de la compilation pour voir si il y a bien une méthode du même nom existant déjà mais il n'est pas obligatoire
@Override
public boolean equals(Object obj) {
    // Si les deux objets ont la même référence, alors elles ont les même valeurs et sont égales
    if (this == obj)
        return true;

    // Si l'objet n'est PAS une instance de la classe Color, alors il ne peut pas être égal
    if (!(obj instanceof Color))
```

```
        return false;

        // Si l'objet est une instance de la classe Color et que tous ses attributs sont égaux, alors les deux objets sont égaux
        Color other = (Color) obj;
        return b == other.b && g == other.g && r == other.r;
    }
```

Ce code (sans les commentaires) a été automatiquement généré par Eclipse (en allant dans `Source` puis `Generate hashCode() and equals()` puis cocher les attributs et la case `Use instanceof to compare types`)

> Petite précision : Dans le code on utilise pas de `else` car quand un `return` arrive, la suite du code n'est pas exécuté donc ce n'est pas nécessaire.

> Deuxième précision : Dans Eclipse, on doit cocher la case pour instanceof sinon Eclipse va utiliser une méthode par défault appellée `.getClass()` et le comportement sera différent. Si une nouvelle classe `MyColor` hérite (prends tous les attributs et méthodes) de `Color`, avec instanceof on peut comparer des objets de `Color` et `MyColor` tandis qu'avec `getClass()` on ne pourra comparer que des `MyColor` ensemble ou des `Color` ensemble.

Redéfinition du .toString()

Le `toString` est utilisé pour avoir une représentation textuelle de l'objet (qui doit être courte et informative).

```
    @Override
    public String toString() {
        return String.format("Color(%d, %d, %d)", r, g, b);
    }
```

Il y a aussi un outil dans Eclipse pour faire cela mais il est merdique et beaucoup plus compliqué que de l'écrire en code directement.

Résumé

Les références :

- Une référence `null` signifie une absence de valeur (une absence d'addresse en mémoire)

Equals et == :

- `==` compare les références et non les valeurs.
- `equals()` sert à comparer les valeurs (propriétés), c'est ce qu'il faut utiliser pour comparer des objets.

L'héritage de Object :

- L'héritage signifie qu'une classe "hérite" de toutes les méthodes et attributs d'une autre
- Toutes les classes par défault dans Java héritent de la classe `Object`
- Il faut donc redéfinir certaines méthodes par défault

Redéfinition de méthodes :

- On peut utiliser le mot clé `@Override` pour signifier que la méthode est une redéfinition d'une autre
- `.equals()` doit être redéfini pour ne pas avoir le même comportement que `==`
- `.toString()` est utilisé pour avoir une courte représentation textuelle d'un objet et doit lui aussi être redéfini

---

Revision #1
Created 27 April 2023 04:31:39 by SnowCode
Updated 27 April 2023 04:35:12 by SnowCode