Héritage (compléments)

Classe finale

Une classe finale est une classe déclarée **final** .

```
public final class Person {
  private String name;
  public Person(String name){this.name = name;}
```

Il n'est pas possible de définir de sous-classes d'une classe `final`.

Méthode finale

Une méthode finale est une méthode déclarée **final** .

```
public class ChessPlayer{
  private String name;
  public ChessPlayer(String name){setName(name);}
  public final String getName() {return name;}
  public final void setName(String name) {this.
  name = name;}
```

Une méthode finale ne peut pas être redéfinie dans une sous-classe.

Il est conseillé de déclarer finale une méthode appelée dans un constructeur. Si un constructeur appelle une méthode non-finale, une sous-classe pourrait redéfinir son comportement pouvant avoir des effets indésirables sur l'objet en cours de création.

La méthode `toString()`

Si vous souhaitez représenter un objet sous forme de chaîne de caractères, pensez à redéfinir la méthode `toString()` héritée de la classe `Object`.

```java
public final class Person {
  //Redéfinition de la méthode toString()
  public String toString(){
    return name;
  }
}
```

Appel implicite

On peut donner la référence d'un objet en créant un `String` par concaténation. Dans ce cas, un représentation de l'objet par un objet `String` est créé en appelant la méthode `toString()` de l'objet.

```
Person p = new Person("John Doe");

System.out.println("Hey " + p); // affiche Hey my
    name is John Doe
```

Attention : Pensez à toujours redéfinir la méthode `toString()`

La méthode héritée `equals`

Une autre méthode héritée de la classe `Object` est la méthode `equals`.

Si vous ne redéfinissez pas cette méthode, sachez qu'elle compare la référence de l'objet avec la référence donnée en argument :

- Si les références sont égales, elle retourne **`true`**
- Si les références sont différentes, elle retourne **`false`**

```
Person p1 = new Person("John␣Doe"), p2 = p1, p3 =
    new Person("John␣Doe");

System.out.println(p1.equals(null)); // false
System.out.println(p1.equals(p2)); // true
System.out.println(p1.equals(p3)); // false
```

Un exemple de redéfinition de la méthode equals

Des objets représentant des points du plan :

```
public class Point {
    private double x, y;

    public Point(double x, double y) {
      this.x = x; this.y = y; }
  }
```

Une première redéfinition de la méthode `equals`

La méthode retourne **`true`** si les états de deux objets `Point` coïncident :

```
    // Redéfinition de la méthode equals()
     public boolean equals(Object o) {
      if (o == this) { return true; }
      if (o == null) { return false; }
      if (getClass() != o.getClass()) {
        return false;}

      Point other = (Point)o; // sous-classement
      return Double.compare(x, other.x) == 0
     && Double.compare(y, other.y) == 0;
     }
}
```

La méthode `getClass()`

La méthode `getClass()` retourne une représentation de la classe ayant servi à créé l'objet par une instance de la classe `Class`. A chaque classe est associé une copie **unique** d'un objet `Class`.

- L'instruction `getClass() == o.getClass()` retourne **`true`** si les deux objets sont du même type.
- L'instruction `getClass() != o.getClass()` retourne **`true`** si les deux objets ne sont pas du même type.

Une remarque importante

> **Attention :** La redéfinition précédente de la méthode `equals` peut ne pas convenir s'il existe une relation d'héritage entre les objets.

Considérons une sous-classe de la classe `Point` :

```
public class PointColore extends Point
{
    private String color;
    public PointColore(double x,double y,String
   color)
    {
      super(x,y); this.color = color;
    }
  }
```

Une remarque importante

La méthode `equals` retournera **false** avec un objet `Point` et un objet `PointColore` même si les deux attributs `x` et `y` coïncident.

```
Point p1 = new Point(1,1),
    p2 = new PointColore(1,1,"blue");

System.out.println(p1.equals(p2)); // affiche false
System.out.println(p2.equals(p1)); // affiche false
```

Une autre redéfinition de la méthode `equals`

Si le comportement précédent n'est pas le comportement recherché, une autre redéfinition de la méthode `equals` doit être considérée :

```
public class Point {
  // Redéfinition alternative de la méthode equals
    ()
    public final boolean equals(Object o) {
     if (o == this) { return true; }
     if (o == null) { return false; }
     if (! o instanceof Point) {
       return false;}

     Point other = (Point)o; // sous-classement
     return Double.compare(x, other.x) == 0
     && Double.compare(y, other.y) == 0;
  }
}
```

Une autre redéfinition de la méthode `equals`

```
Point p1 = new Point(1,1),
    p2 = new PointColore(1,1,"blue");

System.out.println(p1.equals(p2)); // affiche true
System.out.println(p2.equals(p1)); // affiche true
```

Il est conseillé d'ajouter le modificateur **`final`** à la redéfinition de la méthode `equals`.

En résumé

Si vous décidez de redéfinir le comportement de la méthode `equals` héritée de la classe `Object`. Vous aurez à choisir entre une des deux redéfinitions précédentes :

- Si vous utilisez la première version de la redéfinition avec `getClass()`, ayez à l'esprit qu'elle retournera toujours **`false`** avec un objet d'un type `A` et un objet d'un sous-type de `A`.
- Si vous ne recherchez pas le comportement précédent, utilisez la deuxième version mais en n'oubliant pas de la déclarer **`final`** (car il n'y pas de manière correcte de la redéfinir dans ce cas).