Cours de Programmation Orientée Objet

Contrat et application à la librairie standard JAVA

Marc Champesme
mailto:Marc.Champesme@lipn.univ-paris13.fr

29 septembre 2021

Table des matières

Fonctionnalités de la classe Object

- La classe `Object` [1] définie des fonctionnalités communes à toutes les classes (toutes les classes héritent de la classe `Object`).

1. Documentation officielle de la classe `Object` → `http://download.oracle.com/javase/6/docs/api/`

Fonctionnalités de la classe Object

- La classe `Object` [1] définie des fonctionnalités communes à toutes les classes (toutes les classes héritent de la classe `Object`).
- Ces fonctionnalités sont définies par l'intermédiaire de méthodes possédant chacune :

1. Documentation officielle de la classe `Object` → `http://download.oracle.com/javase/6/docs/api/`

Fonctionnalités de la classe Object

- La classe `Object` [1] définie des fonctionnalités communes à toutes les classes (toutes les classes héritent de la classe `Object`).
- Ces fonctionnalités sont définies par l'intermédiaire de méthodes possédant chacune :
  - Un contrat (sous forme de commentaires) précisant le cadre à respecter pour adapter le fonctionnement de ces méthodes à toute nouvelle classe.

1. Documentation officielle de la classe `Object` → `http://download.oracle.com/javase/6/docs/api/`

Fonctionnalités de la classe Object

- La classe `Object` [1] définie des fonctionnalités communes à toutes les classes (toutes les classes héritent de la classe `Object`).
- Ces fonctionnalités sont définies par l'intermédiaire de méthodes possédant chacune :
  - Un contrat (sous forme de commentaires) précisant le cadre à respecter pour adapter le fonctionnement de ces méthodes à toute nouvelle classe.
  - Une implémentation basique utilisable directement par toute nouvelle classe

1. Documentation officielle de la classe `Object` → `http://download.oracle.com/javase/6/docs/api/`

Principales méthodes de la classe Object

- Ce contrat défini en particulier quel doit être le comportement commun pour deux notions très importantes : l'égalité entre objets (i.e. méthode `equals`) et la duplication d'objets (i.e. méthode `clone`).

Principales méthodes de la classe Object

- Ce contrat défini en particulier quel doit être le comportement commun pour deux notions très importantes : l'égalité entre objets (i.e. méthode `equals`) et la duplication d'objets (i.e. méthode `clone`).
- `public boolean equals(Object o)` → égalité entre objets
- `protected Object clone()` → clonage d'objets

Principales méthodes de la classe Object

- Ce contrat défini en particulier quel doit être le comportement commun pour deux notions très importantes : l'égalité entre objets (i.e. méthode `equals`) et la duplication d'objets (i.e. méthode `clone`).
- `public boolean equals(Object o)` → égalité entre objets
- `protected Object clone()` → clonage d'objets
- `public int hashCode()` → pour pouvoir utiliser les objets dans des structures de données utilisant des tables de hashage (par exemple, `java.util.HashSet` ?)
- `public String toString()` → pour un affichage de deboggage.

Contrat de la méthode equals : relation d'équivalence

La documentation officielle (cf. ci-dessous) impose que la méthode `equals` implémente une relation d'équivalence (i.e. réflexive, symmétrique et transitive) :

> *The equals method implements an equivalence relation on non-null object references :*
>
> - *It is reflexive : for any non-null reference value x, x.equals(x) should return true.*
> - *It is symmetric : for any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.*
> - *It is transitive : for any non-null reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.*

Contrat de la méthode equals : cas des références null et consistance

De plus :

- le cas où l'argument est `null` est spécifié :

  For any non-null reference value x, x.equals(null) should return false.

Contrat de la méthode equals : cas des références `null` et consistance

De plus :

- le cas où l'argument est `null` est spécifié :

  For any non-null reference value x, x.equals(null) should return false.

- ainsi que la notion de consistance :

  It is consistent : for any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.

Contrat de la méthode equals : implémentation de la classe Object

Le contrat précise par ailleurs quelle est l'implémentation de base fournie par la classe `Object` :

> *The equals method for class Object implements the most discriminating possible equivalence relation on objects ; that is, for any non-null reference values x and y, this method returns true if and only if x and y refer to the same object (x == y has the value true).*

En d'autres termes, avec cette implémentation, `equals` à la même sémantique que l'opérateur `==`. Il convient donc d'être particulièrement attentif à la nécessité de redéfinir cette méthode pour chaque nouvelle classe.

Contrat de la méthode equals : (fin)

Le contrat donne aussi des indications sur l'incidence de l'implémentation de la méthode `equals` sur l'implémentation de la méthode `hashCode` :

> *Note that it is generally necessary to override the hashCode method whenever this method is overridden, so as to maintain the general contract for the hashCode method, which states that equal objects must have equal hash codes.*

Au delà de la méthode `hashCode`, la façon dont equals est implémentée a aussi une conséquence forte sur l'implémentation de la méthode `clone()` (cf. plus loin).

Retour sur la classe Point : méthode equals

```
public class Point {
    ...
    public boolean equals(Object o) {
        if (!(o instanceof Point)) {
            return false;
        }
        Point autrePoint = (Point) o;
        return getX() == autrePoint.getX()
                && getY() == autrePoint.getY()
    }
    ...
```

Définition du contrat de hashCode dans la classe Object

Returns a hash code value for the object. This method is supported for the benefit of hash tables such as those provided by HashMap.

The general contract of hashCode is :

- *Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.*
- *. . .*

Définition du contrat de hashCode (suite)

Relation avec la méthode equals :

- *If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.*
- *It is not required that if two objects are unequal according to the equals(java.lang.Object) method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.*

Définition du contrat de hashCode (fin)

Implémentation dans la classe `Object` :

> *Implementation Requirements : As far as is reasonably practical, the hashCode method defined by class Object returns distinct integers for distinct objects.*

Définition du contrat de clone dans la classe Object

Creates and returns a copy of this object. The precise meaning of "copy" may depend on the class of the object. The general intent is that, for any object x,

- *the expression : `x.clone() != x` will be true,*
- *and that the expression : `x.clone().getClass() == x.getClass()` will be true, but these are not absolute requirements.*
- *While it is typically the case that : `x.clone().equals(x)` will be true, this is not an absolute requirement.*

By convention, the returned object should be obtained by calling super.clone. If a class and all of its superclasses (except Object) obey this convention, it will be the case that `x.clone().getClass() == x.getClass()`.

Contrat de la méthode clone (suite)

By convention, the object returned by this method should be independent of this object (which is being cloned).

To achieve this independence, it may be necessary to modify one or more fields of the object returned by super.clone before returning it. Typically, this means copying any mutable objects that comprise the internal "deep structure" of the object being cloned and replacing the references to these objects with references to the copies. If a class contains only primitive fields or references to immutable objects, then it is usually the case that no fields in the object returned by super.clone need to be modified.

Implémentation de clone par la classe Object

The method clone for class Object performs a specific cloning operation.

- *First, if the class of this object does not implement the interface Cloneable, then a CloneNotSupportedException is thrown. Note that all arrays are considered to implement the interface Cloneable.*
- *Otherwise, this method creates a new instance of the class of this object and initializes all its fields with exactly the contents of the corresponding fields of this object, as if by assignment ; the contents of the fields are not themselves cloned.*

Thus, this method performs a "shallow copy" of this object, not a "deep copy" operation.

The class Object does not itself implement the interface Cloneable, so calling the clone method on an object whose class is Object will result in throwing an exception at run time.

Règles générales pour le respect du contrat de Object

- Classe non modifiable → toujours redéfinir equals, ne jamais redéfinir clone

Règles générales pour le respect du contrat de Object

- Classe non modifiable → toujours redéfinir equals, ne jamais redéfinir clone
- Classe modifiable → décider de redéfinir equals en fonction des caractéristiques de la classe (décision au cas par cas, pas de règle générale)
- Classe modifiable → si equals est redéfini, clone doit toujours être (re)défini

Règles générales pour le respect du contrat de Object

- Classe non modifiable → toujours redéfinir equals, ne jamais redéfinir clone
- Classe modifiable → décider de redéfinir equals en fonction des caractéristiques de la classe (décision au cas par cas, pas de règle générale)
- Classe modifiable → si equals est redéfini, clone doit toujours être (re)défini
- equals redéfini → redéfinir hashCode
- equals redéfini → redéfinir toString

Retour sur la classe Point : méthode clone

On suppose la classe Point modifiable.

```
public class PointModifiable implements Cloneable {
    ...
    // equals redéfini
    public Object clone() {
        Object leClone = null;
        try {
            leClone = super.clone();
        } catch (CloneNotSupportedException e) {
            throw new InternalError("Erreur impossible"
                                    + "en théorie !");
        }
        // Rien à ajouter: l'implémentation de Object convient
        return leClone;
    }
    ...
```

Table des matières

Les classes conteneur du package java.util

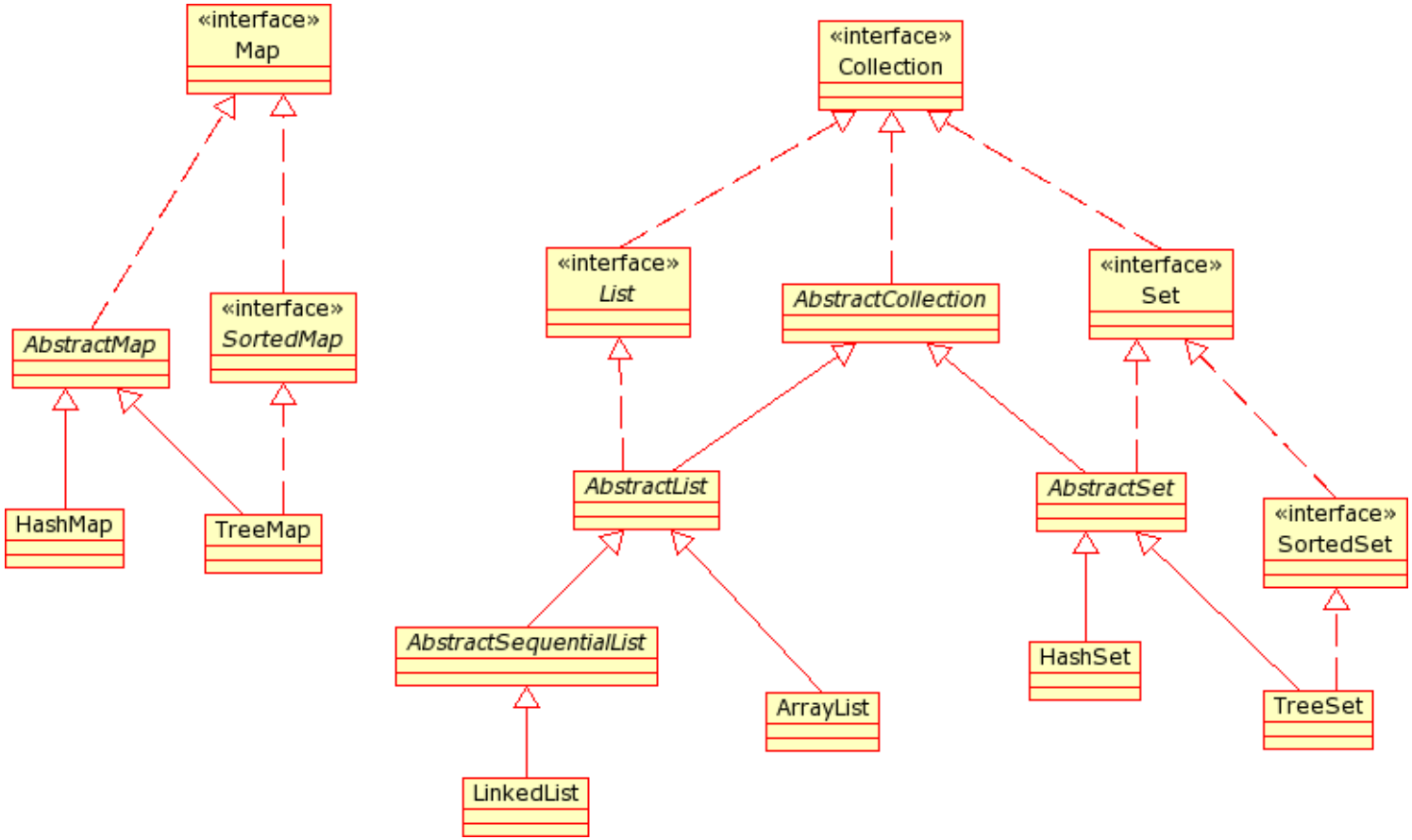

Figure – Hiérarchie des classes conteneur du package java.util

Table des matières

Par des commentaires (format javadoc)

```
/**
 * Un point non modifiable représentant une position dans
 * un espace à deux dimensions. Les coordonnées sont spécifiées
 * par des entiers. Les opérations de transformations affines
 * courantes (homothétie, symétrie, ...) sont définies et
 * retournent des instances de cette classe dont les
 * coordonnées sont arrondies à l'entier le plus proche.
 *
 * @author Marc Champesme
 * @version 1.0 (24 novembre 2006)
 */
public class Point {
    ...
}
```

Commentaires au format javadoc : méthodes

```
...
/**
 * Initialise un point à la position (x, y) du système
 * de coordonnées.
 *
 * @param x abscisse du point
 * @param y ordonnée du point
 */
public Point(int x, int y) {
    this.x = x;
    this.y = y;
}
...
```

Par des assertions (JML)

```
    /** <mêmes commentaires>
     * @invariant ...;
     */
public class Point {
    ...
    /** <mêmes commentaires>
      * @ensures getX() == x;
      * @ensures getY() == y;
      */
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    ...
```

Résumé :

- Classe + masquage d'information (visibilité) $\rightarrow$ possibilité d'associer un contrat rigoureux à chaque classe... et de faire en sorte qu'il soit respecté

Résumé :

- Classe + masquage d'information (visibilité) $\rightarrow$ possibilité d'associer un contrat rigoureux à chaque classe... et de faire en sorte qu'il soit respecté
- Nécessite que le contrat soit spécifié rigoureusement pour chaque classe

Résumé :

- Classe + masquage d'information (visibilité) → possibilité d'associer un contrat rigoureux à chaque classe... et de faire en sorte qu'il soit respecté
- Nécessite que le contrat soit spécifié rigoureusement pour chaque classe
- ... en utilisant des commentaires (spécification informelle)

Résumé :

- Classe + masquage d'information (visibilité) → possibilité d'associer un contrat rigoureux à chaque classe... et de faire en sorte qu'il soit respecté
- Nécessite que le contrat soit spécifié rigoureusement pour chaque classe
- ... en utilisant des commentaires (spécification informelle)
- ... + des assertions (invariant, pré-conditions, post-conditions) → spécification plus formelle (vérifiable)