

JAVA SE – Égalité

L'égalité entre types distincts

Note: le contenu de cette fiche suppose la connaissance de la problématique des ClassLoaders ([voir la fiche correspondante](#)).

La méthode public boolean equals (Object autre) doit être implantée dans chaque classe pour laquelle on souhaite spécifier les critères d'égalité des objets. La documentation de Object précise les conditions que doivent remplir ces codes de calcul de l'égalité (et il est très important de bien les connaître).

Lorsque l'on utilise des outils de développement intégrés (*IDE*) on a des assistants (*wizard*) qui nous proposent des rédactions de code pour cette méthode equals.

Voici un exemple d'un tel code de méthode equals généré automatiquement :

```
public class Point {
    private Integer coorX ;
    private Integer coorY ;
    // autres codes
    public boolean equals(Object obj) {
        if (obj == null) {
            return false;
        }
        if (getClass() != obj.getClass()) {
            return false;
        }
        final Point other = (Point) obj;
        if (this.coorX != other.coorX &&
            (this.coorX == null || !this.coorX.equals(other.coorX))) {
            return false;
        }
        if (this.coorY != other.coorY &&
            (this.coorY == null || !this.coorY.equals(other.coorY))) {
            return false;
        }
        return true;
    }
}
```

Le point qui nous intéresse ici est l'utilisation de getClass() : on rejette toute comparaison avec un objet qui ne serait pas **exactement** du même type. Cette mesure a plusieurs conséquences :

- Si deux ClassLoaders différents chargent ce même code il est possible d'avoir deux instances pourtant égales pour lesquelles le test rend faux (elles ne cohabitent pas dans le même ClassLoader).
- Si on écrit un code qui représente la même donnée mais pour lequel un comportement de méthode est modifié il est impossible d'utiliser l'égalité. Que ce soit par délégation ou par héritage il est tout à fait possible d'écrire un code qui modifie seulement le comportement d'un objet et pas son état: exemple Collections.unmodifiableList(liste) rend une "vue" de liste qui est simplement non modifiable. Il ne serait pas tolérable que la liste et sa vue ne soient pas égales! La même remarque s'applique aux objets mandataires (*proxy, stubs*).
- On ne peut pas comparer un objet d'une classe avec une instance d'une sous-classe. Or fonctionnellement il peut y avoir des cas où c'est possible. Exemple : du point de vue du Point je peux vouloir me comparer à un PointEnCouleur (seul le point de vue des coordonnées nous intéresse).

Remarque critique: il serait tentant de dire que du point de vue du PointEnCouleur les exigences ne sont pas les mêmes. C'est malheureusement impossible : les spécifications de equals exigent une parfaite symétrie.

- On devrait pouvoir comparer des objets totalement différents mais qui ont le même contenu. C'est typiquement le cas des objets qui sont des "containers". Un exemple intéressant est celui de l'interface List: du point de vue de la sémantique de List deux instances sont "égales" si elles contiennent les mêmes objets dans le même ordre.

Il est dans la culture de java de penser à l'égalité entre deux objets de types différents. Dans les codes source de Java il n'est jamais fait usage de getClass() dans les codes de equals. (A l'exception de quelques rares sources dans le package java.text).

Plus significatif encore : lors de la migration vers 1.5 beaucoup de codes du package java.util ont été transformés selon les principes des types paramétrés. Ainsi donc dans la classe ArrayList la méthode add est passée d'une signature add(Object arg) à add(E arg) (E étant le paramètre-type). Pourtant la méthode remove(Object arg) a gardé sa signature. Pourquoi? Parce que tous les codes du package s'appuyant sur equals(Object o) doivent permettre une comparaison avec un autre type que le type-paramètre E! Le code généré par l'assistant n'est pas fondamentalement faux. Il devrait être annoté avec un commentaire

```
// @TODO Evolution equals sans getClass() ?
```

L'assistant permet une première maquette mais il faut envisager de reprendre le code en réfléchissant plus en profondeur.

Exemples :

```
public class Point {
    private Integer coorX ;
    private Integer coorY ;
    // autres codes
    public boolean equals(Object obj) {
        if( ! (obj instanceof Point)) return false ;
        // prend en compte le cas où obj est null
        °°° // suite des comparaisons
    }
}
```

L'ennui ici est que si on permet une égalité avec une sous-classe alors le contrat ne peut être modifié dans le code de la sous-classe (equals pourrait tout aussi bien être marqué final sinon on doit reexprimer les mêmes conditions dans la sous-classe -voir le code de java.awt.Point pour un exemple).

Ce code est tentant mais faux :

```
public class PointEnCouleur{
    private Color couleur;
    // autres codes
    public boolean equals(Object obj) { // FAUX!!!
        if( super.equals(obj)) {
            if(obj instanceof PointEnCouleur) {
                PointEnCouleur autre =
                    (PointEncouleur) obj ;
                // on élimine les cas où
                // les couleurs sont incompatibles
            }
            return true ;
        }
        return false ;
    }
}
```

En effet on pourrait alors avoir : pointA égal à pointANoir et pointABlanc mais pointANoir non égal à pointABlanc (ce qui est rigoureusement interdit, pas les spécifications de equals).

A ce stade on pourrait se demander s'il ne vaudrait pas mieux revenir à l'utilisation de getClass() et faire en sorte qu'un Point ne puisse être égal à un PointEnCouleur. Dans ce cas il faut se poser une question fondamentale : *pourquoi utilisons-nous la technique de l'héritage?*

Note Si la définition de la sous-classe fait perdre une propriété de la super-classe cela implique-t-il que l'utilisation des classes est étanche (pas de polymorphisme)? Dans le cas contraire on va considérer que, fonctionnellement, deux points peuvent avoir les mêmes coordonnées sans être égaux (cela dépend de leur

nature). Il faudra alors être d'une vigilance extrême pour éviter des anomalies de comportement des codes (pour un exemple de souci de consistance entre les propriétés de l'objet et sa méthode equals lire la documentation de l'interface Comparable et analyser les conséquences de l'anomalie résultante dans BigDecimal).

L'intérêt de l'ouverture de l'égalité vers des sous-classes est évident si ces sous-classes sont en fait une autre "vue" sur le même objet (mandataires, etc.).

Un cas plus intéressant est l'utilisation d'une interface avec instanceof :

```
public abstract class AbstractList<E> extends
    AbstractCollection<E> implements List<E> {
    // code standard de la bibliothèque java.util
    public boolean equals(Object o) {
        // implante la sémantique définie dans List
        // pourrait être un code dans un "Trait" (Scala)
        if (o == this) return true;
        if (!(o instanceof List)) return false;
        // ensuite l'autre objet est vu comme "List"
        ...
    }
}
```

On a ici clairement la possibilité de comparer avec un objet d'un type différent.

Bien entendu cette stratégie n'est pas généralisable car on ne dispose pas d'une interface pour toute classe.

Ceci dit cela permet de réaliser des codes qui traversent des frontières de ClassLoader.

Quelques exemples avec des égalités transversales :

- Soit un ClassLoader *tronc* avec deux ClassLoaders subordonnés *brancheA* et *brancheB*.

Soit une interface `com.monbiz.Produit` chargée par *tronc* et une classe `com.monbiz.metier.ProduitAlimentaire` chargée par les deux autres ClassLoaders.

```
public abstract class ProduitAlimentaire implements Produit {
    public boolean equals(Object autre) {
        if (! (autre instanceof Produit)) return false ;
        // ensuite comparaison avec les méthodes de Produit
    }
}
```

Ce code va permettre de comparer deux instances situées dans des ClassLoaders différents.

Par contre si l'interface `Produit` n'est pas chargée par *tronc* mais par les deux autres branches il va falloir passer par un *Proxy* dynamique (`java.lang.reflect.Proxy`) ce qui est particulièrement lourd.

Formations JAVA traitant de ce sujet

Présentation des fondamentaux de la méthode `equals` :

- [USL110 : Java: bases du langage](#) (pour programmeurs débutants)
- [USL275: Java : Programmation pour Développeur Confirmé](#)
- [même programme mais sur 7 jours pour programmeurs débutants](#)

Approfondissements (`equals` et `hashCode`) :

- [IIN43 - mise à niveau et préparation à la certification programmeur](#)

Mises en oeuvre sophistiquées :

- [USL285 : Développement d'application java](#)
- Nos cours "java avancé" adaptés à la demande.

Bernard AMADE

© Demos