

Égalité, ordre et hachage

CS-108

Michel Schinz

2021-03-29

1 Introduction

Comme nous l'avons vu dans le cours sur les collections, les mises en œuvre `TreeSet` et `HashSet` exigent chacune qu'il soit possible d'effectuer certaines opérations sur les éléments de l'ensemble : `TreeSet` doit pouvoir les trier, tandis que `HashSet` doit pouvoir les « hacher ». Il convient d'examiner ces exigences en détail.

Avant cela, il faut toutefois se rendre compte que les ensembles eux-mêmes sont plus exigeants que les listes concernant les éléments qu'ils contiennent. En effet, si les éléments d'une liste peuvent être quelconques, celle-ci se contentant de les stocker, les éléments d'un ensemble doivent au moins être dotés d'une notion d'égalité ! Pourquoi ? Car un ensemble ne peut pas contenir de doublons, or pour déterminer si un élément est un doublon d'un autre, il faut une notion d'égalité.

Commençons donc par examiner la notion d'égalité et sa mise en œuvre dans la bibliothèque Java.

2 Egalité

La notion d'égalité peut sembler évidente au premier abord, mais elle est loin de l'être en pratique.

Pour illustrer les problèmes qu'elle peut poser, considérons deux billets de banque neufs et de même valeur. Sont-ils égaux ? Comme il s'agit de deux billets distincts, portant un numéro de série différent, on pourrait être tenté de répondre que non. Mais il est bien clair que dans la quasi totalité des situations, ils sont parfaitement substituables l'un à l'autre. Dès lors, dans la plupart des cas, il paraît assez logique de les considérer comme égaux.

Comme cet exemple simple l'illustre, il n'y a pas (en général) de notion absolue d'égalité, et déterminer quelle notion d'égalité utiliser dans un contexte donné n'est pas toujours facile.

2.1 Égalité en Java

En programmation, des difficultés similaires à celle de l'exemple ci-dessus se posent. Pour l'illustrer, considérons l'extrait de programme suivant, qui crée deux instances de `Integer`, chacune représentant l'entier 5 :

```
Integer i1 = new Integer(5);  
Integer i2 = new Integer(5);
```

Les objets référencés par `i1` et `i2` sont-ils égaux ou non ? La réponse dépend du point de vue que l'on adopte, comme pour l'exemple des billets de banque.

Les objets référencés par `i1` et `i2` sont des objets distincts, dans le sens où ils occupent une position différente en mémoire. En conséquence, s'il était possible de changer l'entier stocké dans une instance de la classe `Integer` (ce qui n'est pas le cas, cette classe étant immuable), il serait possible de changer le contenu de l'un des deux objets sans que le contenu de l'autre ne soit affecté. De ce point de vue, on peut donc considérer que ces objets ne sont pas égaux.

Cela dit, même si les objets référencés par `i1` et `i2` sont distincts, leur contenu est identique. De cet autre point de vue, on peut donc les considérer comme égaux.

Dès lors, suivant le point de vue adopté, on peut dire que les objets référencés par `i1` et `i2` sont égaux ou non. Les deux points de vue mentionnés ci-dessus correspondent à deux notions d'égalité que l'on retrouve dans beaucoup de langages de programmation, à savoir :

1. l'égalité **par référence** (ou **par identité**), qui considère deux objets égaux si et seulement si il s'agit d'un seul et même objet,
2. l'égalité **par structure** (ou **structurelle**), qui considère deux objets égaux si et seulement si leur contenu est identique.

Dans l'exemple des billets de banque, l'égalité par référence consisterait à les comparer en fonction de leur numéro de série, tandis que l'égalité par structure consisterait à les comparer en fonction de leur valeur monétaire.

L'égalité par référence est la plus discriminante : deux objets égaux par référence sont forcément égaux par structure, mais deux objets égaux par structure ne sont pas forcément égaux par référence. Ainsi, dans l'exemple ci-dessus, les objets référencés par `i1` et `i2` ne sont pas égaux par référence mais le sont par structure.

En Java, l'égalité par référence est prédéfinie sous la forme de l'opérateur `==`. De plus, la méthode `equals` définie dans `Object` — et donc héritée par toute classe qui ne la redéfinit pas — effectue une comparaison par référence.

L'égalité structurelle n'est par contre pas prédéfinie en Java. Pour que les instances d'une classe soient comparées par structure, il faut que celle-ci redéfisse la méthode `equals` et mette en œuvre l'égalité structurelle. De manière générale, il est conseillé de le faire pour les classes immuables, et pour elles seulement, étant donné que l'identité de leurs instances n'a aucune importance.

Règle de l'égalité et de l'immuabilité : Lors de la définition d'une classe immuable, redéfinissez la méthode `equals` afin que ses instances soient comparées par structure. Ne le faites pas lors de la définition d'une classe non immuable.

Cette règle admet une exception importante : s'il est exclu qu'il existe deux instances d'une classe immuable ayant le même contenu — p.ex. deux instances de `Integer` contenant le même entier —, alors l'égalité par référence est équivalente à l'égalité par structure, et il n'est pas nécessaire de redéfinir `equals`.

Notez que, malheureusement, beaucoup de classes de la bibliothèque Java violent la règle de l'égalité et de l'immuabilité, à commencer par toutes les collections, qui ne sont en général pas immuables mais néanmoins comparées par structure.

3 Ordre

Après la notion d'égalité, la seconde notion qu'il convient d'examiner en détail est celle d'ordre. En effet, même s'il n'est pas indispensable de pouvoir ordonner des éléments pour les stocker dans un ensemble (une simple notion d'égalité suffit, comme on l'a vu), cette possibilité permet de fournir une mise en œuvre assez efficace. Intuitivement, cela paraît logique : chercher le nom d'une personne dans une longue liste, par exemple, est beaucoup plus rapide si celle-ci est triée par ordre alphabétique que si elle ne l'est pas.

3.1 Ordre en Java

En Java, il existe deux moyens de spécifier comment ordonner les instances d'une classe donnée :

1. en lui faisant implémenter l'interface `Comparable` et donc fournir une méthode permettant de comparer une instance donnée avec une autre,
2. en utilisant un comparateur, qui est un objet externe à la classe, implémentant l'interface `Comparator` et sachant comparer deux instances de celle-ci.

Ces deux options sont décrites ci-après.

3.2 L'interface `Comparable`

L'interface `Comparable` du paquetage `java.lang` peut être implémentée par toute classe dont les instances sont comparables entre elles. Elle permet d'attacher une notion d'ordre aux instances de la classe en question.

L'interface `Comparable` contient une seule méthode, `compareTo`, qui compare l'objet auquel on l'applique (le récepteur) à un autre objet passé en argument. L'entier retourné par cette méthode est négatif si le récepteur est strictement inférieur à l'argument, nul si les deux sont égaux, et positif sinon.

```
public interface Comparable<T> {
    int compareTo(T that);
}
```

Le fait que l'interface `Comparable` soit générique peut surprendre. A quoi sert son paramètre de type ?

Pour répondre à cette question, il faut constater qu'il donne le type de l'argument de la méthode `compareTo`. Pour comprendre pourquoi cela peut être utile, prenons un cas concret, par exemple celui de la classe `Integer`. De quel type devrait être l'argument la méthode `compareTo` de cette classe ? Logiquement, ce type devrait être `Integer` également, car le seul type d'objets avec lequel il est sensé de comparer une instance de `Integer` est une autre instance de `Integer`...

```
public final class Integer {
    private final int value;

    // ... autres attributs et méthodes

    public int compareTo(Integer that) {
        // ... compare this.value et that.value
    }
}
```

Dès lors, lorsqu'elle implémente l'interface `Comparable`, la classe `Integer` doit lui passer *son propre type* en argument :

```
public final class Integer implements Comparable<Integer> {
    private final int value;

    // ... autres attributs et méthodes

    @Override
    public int compareTo(Integer that) {
        // ... compare this.value et that.value
    }
}
```

De manière générale, toute classe qui implémente l'interface `Comparable` lui passe son propre type en paramètre.

En résumé, la raison pour laquelle l'interface `Comparable` est générique est que cela permet d'assurer que le type de l'argument passé à la méthode `compareTo` est le même que le type du récepteur. Cette utilisation particulière de la généricité est une petite astuce qu'il est bon de connaître, car elle est utile et se rencontre fréquemment en pratique.

3.2.1 Méthodes `compareTo` et `equals`

Une classe implémentant l'interface `Comparable` possède deux manières de tester l'égalité de deux de ses instances :

1. via la méthode `equals` héritée (ou redéfinie) de `Object`,
2. via la méthode `compareTo` de `Comparable`.

Bien entendu, il est important que ces deux méthodes s'accordent sur l'ensemble des objets qui sont égaux, ce qui est énoncé par la règle suivante :

Règle de compatibilité `compareTo` / `equals` : Lorsque vous définissez une classe qui implémente l'interface `Comparable`, assurez-vous que sa méthode `compareTo` soit compatible avec sa méthode `equals`.

Les méthode `compareTo` et `equals` d'une classe donnée sont compatibles ssi elles considèrent exactement les mêmes couples d'objets comme étant égaux, donc si, pour toutes paires d'instances `o1` et `o2` de cette classe, on a :

$$o1.equals(o2) \Leftrightarrow o1.compareTo(o2) == 0$$

3.3 L'interface `Comparator`

Il est évident qu'en Java, une classe donnée ne peut avoir qu'une seule version de la méthode `compareTo`, puisque celle-ci appartient directement à la classe. L'ordre qui est ainsi « attaché » à une classe via sa méthode `compareTo` est appelé **l'ordre naturel** (*natural order*) de cette classe.

Par exemple, l'ordre lexicographique (du dictionnaire) a été choisi par les concepteurs de la classe `String` comme étant l'ordre naturel des chaînes de caractères.

Parfois, plusieurs notions d'ordres peuvent être utiles pour un même type, et il est donc important d'avoir un moyen d'ordonner des valeurs qui soit externe à leur classe. C'est le but de la notion de comparateur.

L'interface `Comparator` du package `java.util` décrit un comparateur, c-à-d un objet capable de comparer deux autres objets d'un type donné. Le paramètre de type de cette interface spécifie le type des objets que le comparateur sait comparer :

```
public interface Comparator<T> {  
    int compare(T o1, T o2);  
}
```

La méthode `compare` doit retourner un entier négatif si le premier objet est inférieur au second, nul si les deux sont égaux et positif dans les autres cas.

3.4 Comparable et Comparator

Les interfaces `Comparable` et `Comparator` ont un rôle similaire mais sont néanmoins différentes : l'interface `Comparable` est implémentée par les objets comparables eux-mêmes, alors que l'interface `Comparator` est implémenté par un objet étranger. Dès lors :

- la méthode `compareTo` de `Comparable` prend *un seul* argument, et compare le récepteur (`this`) avec cet argument,
- la méthode `compare` de `Comparator` prend *deux* arguments, et les compare entre eux.

Par exemple, en Java, la classe `Integer` implémente l'interface `Comparable`. Cela signifie que les entiers « savent » se comparer entre eux. Cette notion d'ordre, directement attachée aux entiers, est leur ordre naturel et correspond à l'ordre habituel en mathématiques.

Il est néanmoins possible de définir d'autres ordres sur les entiers, par exemple pour inverser l'ordre naturel. Pour ce faire, on définit un comparateur, qui est un objet séparé des entiers eux-mêmes, qui implémente l'interface `Comparator` et qui compare les entiers selon l'inverse de l'ordre naturel.

On peut illustrer l'utilité d'avoir à la fois l'interface `Comparable` et l'interface `Comparator` au moyen de la méthode `sort` de l'interface `List`. Pour mémoire, cette méthode prend en argument le comparateur à utiliser pour trier les éléments de la liste, mais si ce comparateur est nul, l'ordre naturel des éléments est utilisé.

L'extrait de programme ci-dessous utilise ces deux options pour trier une liste d'entiers d'abord dans l'ordre naturel des entiers (donc sans fournir de comparateur), puis dans l'ordre inverse, en utilisant un comparateur inversant :

```
List<Integer> l = Arrays.asList(1, 3, 5, 7, 2, 4, 6, 8);

l.sort(null);           // utilise l'ordre naturel (1 < 2 < ...)
System.out.println(l); // imprime [1, 2, 3, 4, 5, 6, 7, 8]

l.sort((i, j) -> Integer.compare(j, i));
System.out.println(l); // imprime [8, 7, 6, 5, 4, 3, 2, 1]
```

3.5 TreeSet

La classe `TreeSet` met en œuvre les ensembles d'éléments en les triant, afin de pouvoir y accéder plus rapidement. Une conséquence intéressante de cette organisation est que, lors d'un parcours, les éléments sont toujours fournis triés en ordre croissant.

Par défaut, une instance de `TreeSet` utilise l'ordre naturel des éléments qu'elle contient pour les trier. Ainsi, l'extrait de programme suivant affiche les éléments de l'ensemble construit — les entiers de 1 à 8 — dans l'ordre croissant :

```
Set<Integer> s = new TreeSet<>();
s.addAll(Set.of(1, 3, 5, 7, 2, 4, 6, 8));
for (int i: s)
    System.out.println(i); // imprime 1, puis 2, ..., puis 8
```

Il est également possible de créer une instance de `TreeSet` triant ses éléments selon l'ordre d'un comparateur donné explicitement. Celui-ci doit être passé lors de la construction de l'ensemble. La variante ci-dessous de l'extrait de programme précédent passe un comparateur qui trie les éléments de l'ensemble du plus grand au plus petit, et ils sont donc affichés dans cet ordre :

```
Set<Integer> s = new TreeSet<>((i,j) -> Integer.compare(j,i));
s.addAll(Set.of(1, 3, 5, 7, 2, 4, 6, 8));
for (int i: s)
    System.out.println(i); // imprime 8, puis 7, ..., puis 1
```

4 Hachage

Après les notions d'égalité et d'ordre, il convient d'examiner une troisième notion très utile (entre autres) dans la mise en œuvre des ensembles : le hachage. Cette notion permet de mettre en œuvre les ensembles de manière plus efficace encore que l'ordre, mais pour des raisons peut-être moins faciles à comprendre intuitivement.

4.1 Hachage

Le **hachage** (*hashing*) consiste à transformer une donnée quelconque en un entier, généralement compris dans un intervalle borné. Cette transformation est faite par une **fonction de hachage** (*hash function*) qui, appliquée à une donnée, produit l'entier qui lui correspond, que l'on nomme sa **valeur de hachage** (*hash value*).

Un exemple de fonction de hachage sur les chaînes de caractères composées uniquement des 26 lettres non accentuées de l'alphabet latin est :

$$h(c) = \sum_{i=1, \dots, n} \text{pos}(c_i) \bmod 100$$

où `pos` retourne la position d'une lettre dans l'alphabet — `a` étant à la position 1, `b` à la position 2, etc. Par exemple :

$$\begin{aligned} h(\text{chien}) &= (3 + 8 + 9 + 5 + 14) \bmod 100 = 39 \\ h(\text{niche}) &= (14 + 9 + 3 + 8 + 5) \bmod 100 = 39 \\ h(\text{zoologie}) &= (26 + 15 + 15 + 12 + 15 + 7 + 9 + 5) \bmod 100 = 4 \end{aligned}$$

Le hachage est une technique fondamentale en informatique qui rencontre de nombreuses applications, p.ex. en cryptographie. Ici, nous n'examinerons que son utilisation dans le cadre des collections.

Une fonction de hachage est une fonction mathématique, au sens où lorsqu'on l'applique à deux valeurs égales, elle produit la même valeur de hachage. En d'autres termes, toute fonction de hachage h satisfait la propriété suivante :

$$\forall x, y : x = y \Rightarrow h(x) = h(y)$$

L'implication inverse n'est vraie que pour le cas — très rare en pratique — des fonctions de hachage parfaites.

Une fonction de hachage h est dite **parfaite** si elle fait correspondre une valeur de hachage différente à chaque donnée à laquelle on peut l'appliquer, c-à-d si :

$$\forall x, y : x \neq y \Rightarrow h(x) \neq h(y)$$

ce qui est équivalent à :

$$\forall x, y : h(x) = h(y) \Rightarrow x = y$$

En termes mathématiques, une fonction de hachage est parfaite ssi elle est injective.

En pratique, cette condition n'est que rarement satisfaite. Selon le principe des tiroirs (*pigeonhole principle*), elle ne peut d'ailleurs pas l'être si le nombre de données possibles est plus grand que le nombre de valeurs de hachage possibles, ce qui est fréquemment le cas.

Les fonctions de hachage ne pouvant généralement pas être parfaites, elle doivent être conçues pour distribuer aussi bien que possible les valeurs rencontrées en pratique. Cette condition est malheureusement très difficile à spécifier et à garantir, la définition de bonnes fonctions de hachage étant plus un art qu'une science.

Lorsque deux données distinctes possèdent la même valeur de hachage, c-à-d lorsque $x \neq y$ mais $h(x) = h(y)$, on dit qu'il y a **collision de hachage** (*hashing collision*).

4.2 Hachage en Java

Les concepteurs de Java ont fait le choix — discutable — de fournir dans la classe `Object` la méthode `hashCode` qui retourne, sous la forme d'un entier de type `int`, une valeur de hachage de l'objet auquel on l'applique.

La mise en œuvre par défaut de cette méthode — héritée par toute classe qui ne la redéfinit pas — retourne une valeur qui dépend de l'identité de l'objet. Par défaut, deux objets distincts ont donc *généralement* une valeur de hachage distincte. Attention toutefois : cela n'est pas garanti !

Il est absolument fondamental que deux objets considérés comme égaux par `equals` aient la même valeur de hachage d'après `hashCode`, faute de quoi `hashCode` ne définit pas une fonction de hachage au sens mathématique. En d'autres termes, la propriété suivante doit être satisfaite pour toute paire d'objets x et y :

`x.equals(y) ⇒ x.hashCode() == y.hashCode()`

Les méthodes `hashCode` et `equals` d'une classe donnée sont dites **compatibles** si cette condition est satisfaite.

Règle de compatibilité `hashCode` / `equals` : Si vous redéfinissez `hashCode` dans une classe, redéfinissez également `equals` — et inversement — afin que ces deux méthodes restent compatibles.

L'écriture de fonctions de hachage de qualité étant très difficile, il est préférable de laisser cette tâche à des spécialistes. Heureusement, depuis peu la bibliothèque Java offre dans la classe `Objects` (avec un *s*!) du paquetage `java.util` la méthode (statique) `hash` permettant de calculer une valeur de hachage pour une combinaison arbitraire d'objets :

- `int hash(Object... values)`

Règle de mise en œuvre de `hashCode` : Lorsque vous redéfinissez `hashCode`, utilisez la méthode statique `hash` de la classe `Objects` pour la mettre en œuvre, en lui passant tous les attributs à hacher.

Cette règle admet quelques exceptions, présentées plus bas.

Par exemple, pour une classe représentant une personne, la méthode `hashCode` pourrait être redéfinie ainsi :

```
public final class Person {
    private final String firstName, lastName;
    private final Date birthDate;

    // ... constructeur et autres méthodes

    @Override
    public int hashCode() {
        return Objects.hash(firstName, lastName, birthDate);
    }
}
```

La règle ci-dessus admet quelques exceptions dans des cas particuliers :

1. une classe ne possédant qu'un seul attribut peut utiliser la valeur de hachage de cet attribut comme sa propre valeur de hachage, sans passer par la méthode `hash`,
2. une classe ne possédant qu'un seul attribut de type entier, p.ex. `int`, peut directement utiliser sa valeur comme valeur de hachage.

4.3 HashSet

La classe `HashSet` met en œuvre les ensembles d'éléments en les distribuant en mémoire selon leur valeur de hachage, afin de pouvoir y accéder plus rapidement. Une conséquence de cette technique de mise en œuvre est que l'ordre dans lequel les éléments d'un tel ensemble est parcouru est quelconque.

5 Résumé

Les exigences placées par les listes et les ensembles (et leurs mises en œuvre) en Java sont résumées par la table ci-dessous. Pour chaque type de collection, elle montre les méthodes qui doivent être fournies (correctement !) par les éléments pour qu'on puisse les stocker dans la collection en question. Etant donné que `TreeSet` offre deux possibilités pour déterminer l'ordre des éléments, cette collection apparaît deux fois dans la table.

Collection	Méthodes requises
<code>List<E></code>	<i>aucune</i>
<code>Set<E></code>	<code>equals</code>
<code>HashSet<E></code>	<code>equals</code> et <code>hashCode</code>
<code>TreeSet<E></code>	<code>equals</code> et <code>compareTo</code>
<code>TreeSet<E></code>	<code>equals</code> et <code>compare</code>

6 Références

- *Effective Java (3rd ed.)* de Joshua Bloch, en particulier :
 - la règle 10, *Obey the general contract when overriding equals* sur la redéfinition de la méthode `equals`.
 - la règle 11, *Always override hashCode when you override equals* sur la compatibilité des méthodes `hashCode` et `equals`.