Chapitre 10

Les interfaces Comparable et Comparator[1]

[1] Ce chapitre a été extrait du document "Objets, Algorithmes, Patterns" de [René Lalement], chapitre « Patterns, Les relation d'ordre » http://cermics.enpc.fr/polys/oap/node106.html Il a été complété par des commentaires supplémentaires.

Les relations d'ordre

Plusieurs structures de données (notamment SortedSet et SortedMap) et algorithmes sur celles-ci (tri d'une collection ou d'un tableau, recherche binaire dans un tableau trié) supposent que leurs éléments peuvent être comparés par une relation d'ordre.

Java exprime cette hypothèse en demandant que la classe des éléments implémente l'interface Comparable ; c'est le cas d'un certain nombre de classes usuelles, dont les éléments sont comparables par un ordre usuel, par exemple :

Byte, **Short**, **Integer**, **Long**, **Float**, **Double**, **BigInteger** et **BigDecimal**: Ordre numérique signé.

Char (caractère): Ordre numérique non signé (donc alphabétique).

File: ordre lexicographique sur le chemin d'accès.

String: ordre lexicographique.

Date: Ordre chronologique.

Rappelons que l'interface Comparable déclare une seule méthode

```
interface Comparable {
     int compareTo(Object o);
}
```

Cette méthode retourne un entier <0, nul ou >0 selon que l'objet auquel elle est appliquée précède, est égal ou suit o.

Voici un exemple de classe qui implémente cette interface ; le constructeur vérifie que ses deux arguments sont non nuls, et déclenche l'exception **`NullPointerException`** si ce n'est pas le cas, afin de garantir que les méthodes qui s'appliqueront aux champs prénom et nom ne déclencheront pas cette exception

```
import java.util.*;


class Nom implements Comparable {
      private String prénom, nom;

      String valNom() {return nom;}
      String valPrénom() {return prénom;}

      public Nom(String prénom, String nom) {
            if (prénom==null || nom==null)

                  throw new NullPointerException();
            this.prénom = prénom;
            this.nom = nom;
      }
```

On notera que les méthodes **`equals()`** et **`compareTo()`** se comportent différemment si l'objet n'a pas le type requis, ici Nom : le test **`o instanceof Nom`** permet à **`equals(Object)`** de retourner false, tandis que **`compareTo(Object)`**, ne procédant pas à ce test, peut déclencher l'exception **`ClassCastException`** due au transtypage **`(Nom) o`**.

D'autre part, toute classe qui redéfinit `equals()` doit aussi redéfinir **`hashCode()`**. En effet, deux objets égaux par **`equals()`** doivent avoir la même valeur de hachage par **`hashCode()`**. Ces contraintes (sur **`equals()`**, **`compareTo()`**, **`hashCode()`**, etc.) doivent être respectées, afin d'assurer à l'utilisateur que les méthodes qui les utilisent (par exemple, Collections.sort, etc.) font bien ce qu'elles sont sensées faire.

```
      public boolean equals(Object o) {
            return
                  o instanceof Nom &&
                  ((Nom)o).prénom.equals(prénom) &&
                  ((Nom)o).nom.equals(nom);
      }

      public int hashCode() {
            return 31*prénom.hashCode() + nom.hashCode();
      }

      public int compareTo(Object o) {
            Nom n = (Nom)o;
            int compNom = nom.compareTo(n.nom);
            return
                  compNom!=0 ? compNom : prénom.compareTo(n.prénom);
      }

      public String toString() {
            return prénom + " " + nom;
      }
}
```


Enfin, il arrive que des données doivent être comparées selon plusieurs relations : parfois, selon le nom, parfois selon le prénom, etc. Dans ce cas, associer à la classe un ordre naturel en lui faisant implémenter l'interface Comparable n'est pas suffisant. L'API offre une autre interface, **`Comparator`**, à cette fin.

Ajouter à cela, si vous avez besoin d'ajouter un autre critère de comparaison sachant que le **`compareTo`** est déjà utilisé, vous ne pouvez pas donc l'implémenter une deuxième fois. L'interface comparable vient à la rescousse.

```
interface Comparator {
      public int   compare(Object o1, Object o2);
      public boolean equals(Object obj);
}
```

L'exemple qui suit montre un exemple d'implémentation de cette interface.

Il n'est pas nécessaire de définir dans cet exemple la méthode **`equals`**. En effet nous avons récupéré la méthode **`equals`** déjà définie dans la classe **`Object`**. Il est utile de définir **`equals`** (donc redéfinir la méthode de la classe **`Object`**) que si vous jugez que votre méthode (**`equals`**) est plus efficace et donc rapide.

```
class PrénomComparator implements Comparator {
      public int compare(Object o1, Object o2) {
            Nom r1 = (Nom) o1;
            Nom r2 = (Nom) o2;
            int prénomComp =
                  r1.valPrénom().compareTo(r2.valPrénom());
            if (prénomComp != 0)
                  return prénomComp;
            else
                  return r1.valNom().compareTo(r2.valNom());
      }
}
```

On peut alors créer un objet comparateur, et le passer en argument à certaines méthodes qui l'utilisent, par exemple la fonction de tri **`Collections.sort()`** :

```
class TestcmpTo{
     public static void main(String[] args) {
          Comparator prénomComparator = new PrénomComparator();
          List l = new ArrayList();
          l.add(new Nom("Jacque","Brel"));
          l.add(new Nom("Jackie","Stewart"));
          l.add(new Nom("Jackie","Chan"));
```

Prise en compte du compareTo uniquement. Nous allons donc considéré le tri suivant l'ordre naturel. En cas d'égalité, nous allons trier en fonction du prénom.

```
          Collections.sort(l);
          System.out.println("Comparable: " + l);
```

Prise en compte du compare uniquement. Nous allons donc considérer le tri suivant d'abord le prénom, puis le nom.

```
          Collections.sort(l, prénomComparator);
          System.out.println("Comparator: " + l);

     }
}
```

Affichage en sortie

```
Comparable: [Jacque Brel, Jackie Chan, Jackie Stewart]
Comparator: [Jackie Chan, Jackie Stewart, Jacque Brel]
```