

# TP Collections (2) - sorted sets

Marianne Simonot

## 1 La Structure de données SortedSet

### Aide-mémoire

Beaucoup plus utile encore que les `set`, il y a en java une structure appelée `SortedSet`.

**Définition** Un `SortedSet` est une structure qui permet comme un `Set` de ne pas avoir de doublons et en plus ordonne ses éléments. Pour qu'il puisse réaliser ces 2 choses, on doit lui fournir une relation de comparaison.

**SortedSet est une interface** Pour créer des objets de type `SortedSet`, il faut utiliser l'une de classes Java implémentant cette interface par exemple `TreeSet`.

**Quelques méthodes de SortedSet** Comme sur les `ArrayList` et les `Set` nous avons entre autre `add`, `remove` `contains` et la boucle `foreach`.

Contrairement aux `ArrayList` et comme les `Set`, nous n'avons aucune des méthodes permettant d'accéder aux éléments par leur indice dans la liste : `get(i)` `add(i,e)`, `remove(i)` ne sont pas définies sur les `set`. Il est donc impossible de parcourir un `set` au moyen d'une boucle `for (int i=0; i<l.size;i++)....`

### exercice1

Cet exercice permet de comprendre les relations de comparaison et comment le `SortedSet` les utilise.

Supposons qu'on ait 5 produits et une méthode `compare` définis comme suit

```
public class Produit {  
    private int reference;  
    private int prix;
```

```
    public Produit(int reference, int prix) {  
        super();  
        this.reference = reference;  
        this.prix = prix;
```

```

}

public int getReference() {
    return reference;
}

public int getPrix() {
    return prix;
}

```

```

Produit p1,p2,p3,p4,p5;
p112et30= new Produit(112, 30);
p150et15= new Produit(150, 15);
p120et30= new Produit(120,30);
p112et40 = new Produit(112, 40);
p120et30= new Produit(120,30);
// *****

public int compare(Produit p, Produit q) {
    if (p.getReference() == q.getReference()) {
        return p.getPrix() - q.getPrix();
    } else
        return p.getReference() - q.getReference();
}

```

1.  $\otimes$  (activité papier) Utilisez les règles qui suivent pour ranger les 5 produits dans  $c$  puis complétez les phrases.

### Les règles

- Si  $\text{compare}(p,q) > 0$  alors  $p$  doit être après  $q$  dans  $c$ .
- Si  $\text{compare}(p,q) < 0$  alors  $p$  doit être avant  $q$  dans  $c$ .
- Il n'y a jamais dans  $c$  2 produits  $p,q$  tel que  $\text{compare}(p,q) = 0$

$c = \{$    $\}$

$c$  est ordonnée par ordre

On ne conserve qu'un exemplaire des produits qui ont même

2.  $\oplus$  Même question mais avec cette nouvelle définition de **compare** :

```

public int compare(Produit p, Produit q) {
    if (p.getPrix() == q.getPrix()) {
        return p.getReference() - q.getReference();
    } else
        return p.getPrix() - q.getPrix();
}

```

$c = \{$    $\}$

$c$  est ordonnée par ordre

On ne conserve qu'un exemplaire des produits qui ont même

3.  $\otimes$  Même question mais avec cette nouvelle définition de **compare** :

```

public int compare(Produit p, Produit q) {
    return p.getPrix() - q.getPrix();
}

```

$c = \{$    $\}$

$c$  est ordonnée par ordre

On ne conserve qu'un exemplaire des produits qui ont même

## Aide-mémoire

L'exercice précédent vous a familiarisé avec la méthode `compare` et devrait vous avoir convaincu qu'elle donne toutes les indications permettant de construire une collection d'éléments ordonnés et sans répétition.

Cela se passe comme cela en Java : pour créer un `TreeSet` (implémentation de `SortedSet`), il faut transmettre une méthode `compare`. Comme dans un langage objet les paramètres ne peuvent pas être des méthodes, on va transmettre un objet qui sait comparer, un comparateur.

**Les comparateurs** Un comparateur est un objet instance d'une classe implémentant l'interface fonctionnelle suivante (E désigne ici une classe quelconque) :

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

`compare` doit être implémentée de façon à :

- retourner un entier négatif si o1 est plus petit que o2,
- retourner un entier positif si o1 est plus grand que o2
- retourner 0 si o1 est égal à o2.

`compare` sera utilisée pour gérer les doublons : si `compare(a1,a2) == 0` et que a1 est déjà dans le `sortedSet`, alors a2 ne sera pas ajouté.

`compare` sera utilisée pour ranger les éléments : si on a ajouté a1 et a2 dans un set et si `compare(a1,a2) < 0` alors a1 sera avant a2.

**Le constructeur de TreeSet** Le constructeur de `TreeSet` est de signature :  
`public TreeSet(Comparator<E> c)`

## exercice 2 (\*)

1. On va maintenant créer pas à pas un `SortedSet` de produit ordonné par référence puis par prix en pur Objet.
  - (a) Ecrire une classe `CompareurParRefPuisPrix` qui Implémente l'interface `Comparator`. Le code de la méthode `compare` doit être celui vu dans l'exercice précédent.
  - (b) Téléchargez la classe `LancementProduit` et ajouter dans le main la déclaration d'un objet de type `CompareurParRefPuisPrix`. Utilisez le pour comparer les produits `p120et30` et `p112et40` et vérifiez qu'il vous retourne un entier positif (ce qui signifie que le premier est le plus grand).
  - (c) Créez un `SortedSet` en recopiant l'instruction suivante et en remplaçant les ??? par votre objet comparateur. `SortedSet<Produit> ss1 = new TreeSet<>(????? );`

- (d) Ajouter au `SortedSet ss1` les 5 produits dans n'importe quel ordre et vérifiez en l'affichant que `ss1` a bien 4 éléments et qu'ils sont ordonnés par référence puis par prix.
2. Comme `Comparator` est une interface fonctionnelle, on peut donner le comparateur en utilisant une expression lambda. Refaites le travail en utilisant une expression lambda.

### exercice 3

1. Créer une classe `Gens`. Les gens ont un prénom et un âge.
2. Dans une classe `LancementGens`, déclarer 4 personnes : lulu 18 ans, toto 17 ans, lulu 20 ans, bibi 25 ans.
3. Créez un `SortedSet<Gens>` contenant ces 4 personnes. On veut qu'ils soient ordonnés par ordre des noms puis par âge.  
Pour savoir comment ordonner des chaînes de caractères, consultez l'aide mémoire qui suit.

#### Aide-mémoire

**l'ordre naturel sur les `String`** La classe `String` contient une méthode `public int compareTo(String other)` permettant de comparer des chaînes selon l'ordre du dictionnaire (ordre lexicographique).

`''avion''.compareTo(''bebe'')` retourne un entier positif car avion est avant bebe dans le dictionnaire.

`''avion''.compareTo(''art'')` retourne un entier négatif car avion est après art dans le dictionnaire.

`''avion''.compareTo(''avion'')` retourne 0.