# Les collections (suite)

*Recherche et tri des objets*

## Sommaire



Mohamed Ramzi HADDAD

# Introduction

Dès sa version 1.2, Java propose différents types de collections permettant de regrouper plusieurs objets dans une même structure de données. Chaque type de collection propose ses propres méthodes, conventions et contraintes afin de répondre à tous les besoins. Ceci dit, les collections (Map, List, Queue et Set) sont un moyen simple, performant et élégant pour la manipulation des ensembles d'objets.

Dans ce contexte, les collections peuvent contenir indifféremment des objets instances de classes standards prédéfinies que de classes nouvellement définies par un programmeur. En d'autres termes, les instances de n'importe quelle classe implantée en Java peuvent être stockées dans une collection. Par contre, les collections d'objets instances de classes non prédéfinies nécessitent l'intervention du programmeur pour pouvoir être utilisées correctement.

# La notion d'égalité et la recherche d'objets complexes

Examinons les sorties du programme suivant :

```
public class Personne {
	String nom;
	String prenom;
	int age;
	public Personne(String nom, String prenom, int age) {
		super();
		this.nom = nom;
		this.prenom = prenom;
		this.age = age;
	}
}
```

```
public static void main(String[] args) {
List<Personne> liste = new ArrayList<Personne>();
Personne p1 = new Personne("nom1", "prenom1", 25);
Personne p2 = new Personne("nom2", "prenom2", 28);
liste.add(p1);
liste.add(p2);
System.out.println("voici la liste : "+liste);
Personne p3 = new Personne("nom2", "prenom2", 28);
System.out.println("Recherche de "+p3+" : "+liste.contains(p3));
liste.remove(p3);
System.out.println("voici la liste : "+liste);
}
```

Résultat :

```
voici la liste : [[prenom1 nom1:25], [prenom2 nom2:28]]
Recherche de [prenom2 nom2:28] : false
voici la liste : [[prenom1 nom1:25], [prenom2 nom2:28]]
```

Remarquons ici que malgré que l'objet p2 et p3 sont identiques (de point de vue valeur des attributs), ils ne le sont pas par rapport à la méthode de recherche et par conséquence par la méthode de suppression.

Ceci est dû au fait que le test d'égalité repose sur la comparaison des adresses de objets et non pas de leurs valeurs. En effet les méthodes citées font usage de la méthode **equals** de la classe **Object**¸ celle-ci retourne vrai quand les objets comparés représentent la même instance et donc la même adresse en mémoire. Le code suivant prouve ces faits :

```
System.out.println(p2.equals(p3));
System.out.println(p2.equals(p2));
```

Résultat:

```
false
true
```

Donc, pour palier à ce problème, il est nécessaire de redéfinir la méthode **equals** dans les classes dérivées de la classe **Object** à chaque fois qu'un test d'égalité est prévu. Cette méthode a la signature suivante :

```
public boolean equals(Object obj);
```

La méthode **equals** prend comme paramètre un objet et le compare à celui dont elle fait partie. Pour la classe personne, la redéfinition de la méthode **equals** est la suivante. Nous supposons ici que deux personnes ayant les mêmes valeurs nom, prénom et âge sont égales.

```
public boolean equals(Object obj) {
Personne p = (Personne)obj;
return((this.nom.equals(p.nom))&&(this.prenom.equals(p.prenom))&&(thi
s.age==p.age));
}
```

Examinons maintenant la sortie du programme précédent :

```
voici la liste : [[prenom1 nom1:25], [prenom2 nom2:28]]
Recherche de [prenom2 nom2:28] : true
voici la liste : [[prenom1 nom1:25]]
p2.equals(p3) : true
p2.equals(p2) : true
```

La redéfinition de la méthode **equals** dépend de la sémantique des objets et des besoins du programmeur. Il est donc possible de considérer que deux objets sont égaux s'ils ne partagent qu'un sous ensemble des valeurs de leurs attributs. Dans certains cas, il n'est pas nécessaire d'effectuer cette redéfinition si chaque objet devrait être considéré comme unique.

Dans ce cas, il est toujours possible de tester si deux variables font référence au même objet en utilisant l'opérateur de comparaison **==**. En effet la comparaison de deux personnes identiques avec cet opérateur donne :

```
p2 == p3 : false
```

# La notion d'ordre et la comparaison des objets complexes

## 2.1 Les types comparables

Examinons le code suivant utilisant la classe personne précédemment sus indiquée :

```
public static void main(String[] args) {
	List<Personne> liste = new ArrayList<Personne>();
	Personne p1 = new Personne("nom1", "prenom1", 29);
	Personne p3 = new Personne("nom2", "prenom2", 25);
	Personne p2 = new Personne("nom2", "prenom2", 28);
	liste.add(p1);
	liste.add(p2);
	liste.add(p3);
	System.out.println("liste initiale : "+liste);
	Collections.sort(liste);
	System.out.println("liste triée : "+liste);
}
```

Bound mismatch: The generic method sort(List<T>) of type Collections is not applicable for the arguments (List<Personne>). The inferred type Personne is not a valid substitute for the bounded parameter <T extends Comparable<? super T>>

Le code présente une erreur lors de sa compilation indiquant que les éléments de la liste ne sont pas comparables. Ceci est attendu puisque nous n'avons fournis à la méthode **sort** ni le moyen de comparer les personnes, ni la définition de la notion d'ordre entre elles. En effet, la méthode **sort**, qui prend qu'un seul paramètre et qui trie la collection dans un ordre croissant, a la signature suivante :

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

Ceci implique que les éléments de la liste doivent implémenter l'interface comparable. Celle-ci ne contient qu'une seule méthode abstraite nommée compareTo qui permet de comparer l'objet sur lequel elle est invoquée et celui qu'elle prend en paramètre. De cette manière les objets d'une classe qui implémente cette interface deviennent mutuellement comparables. Voici le code à rajouter à la classe Personne pour corriger le code précédent.

```
public class Personne implements Comparable<Personne>
```

L'implémentation de l'interface comparable requière la définition de la méthode compareTo. Celle-ci compare deux objets et est invoquée comme suit :

```
int x = objetCourant.compareTo(unAutreObjet);
```

compareTo retourne :

- une valeur négative si objetCourant est inférieur au sens de la comparaison à l'autre objet,
- zéro si les deux objets sont égaux, et
- une valeur positive si objetCourant est supérieur à l'objet `unAutreObjet`.

Voici donc un exemple d'implémentation de la méthode compareTo pour la classe personne et dans lequel nous supposons qu'une personne précède une autre si elles est moins agée :

```
public int compareTo(Personne o) { return this.age - o.age; }
```

Dans ce cas, sachant que la méthode **sort** trie les éléments dans l'ordre croissant au sens de la comparaison (c'est-à-dire que si la comparaison d'un élément avec son suivant donne une valeur positive, ils seront permutés), le tri de la liste ordonnera les personnes selon l'ordre croissant de leurs âges. Voici le résultat de l'exemple précédent :

```
liste initiale: [[prenom1 nom1:29], [prenom2 nom2:28], [prenom2 nom2:25]]
liste triée: [[prenom2 nom2:25], [prenom2 nom2:28], [prenom1 nom1:29]]
```

Attention à l'implémentation de la méthode compareTo lors de la spécification de la valeur de retour. En effet, l'implémentation suivante donne que les personnes seront triées dans le sens décroissant.

```
public int compareTo(Personne o) { return o.age  - this.age; }
```

## 2.2 Les comparateurs

Dans la réalité, il est possible d'avoir besoin d'ordonner les personnes de différente manière (par rapport au nom, au prénom, à l'âge, au salaire, etc…), mais il n'y a qu'une seule méthode compareTo . Dans ces cas, il est nécessaire d'avoir recours à la deuxième version surchargée de la méthode **sort** et qui prend comme paramètres la collection à trier ainsi qu'un comparateur. Voici la signature de la méthode **sort** :

```
public static <T> void sort(List<T> list, Comparator<? super T> c)
```

L'interface **Comparator** donne au programmeur le moyen de comparer et d'introduire une notion d'ordre entre les objets d'une classe. Parmi les avantages de cette classe, citons :

- il est désormais possible de définir plusieurs comparateurs pour la même classe.
- il devient possible de définir un ordre non pas seulement entre les objets des classes nouvellement définies mais aussi pour celles qui sont déjà prédéfinies dans Java (String, Integer, etc…).
- Il n'est plus nécessaire de modifier le code des classes à trier ou à comparer pour introduire la méthode compareTo et implémenter l'interface Comparable.

Le recours à l'interface **Comparator** est simple ; il suffit de l'implémenter et de définir la méthode compare. Voici un exemple de comparateur de personne, utilisant l'âge comme critère de comparaison.

```
public class TriPersonneSelonAge implements Comparator<Personne> {
	public int compare(Personne p1, Personne p2) {
		return p1.age - p2.age;
	}
}
```

Enfin la ligne suivante permet de trier la liste des personnes selon l' ordre croissant des âges:

```
TriPersonneSelonAge triAge = new TriPersonneSelonAge();
Collections.sort(liste,triAge);
```

La classe **String** implémente l'interface **Comparable**, donc elle propose la méthode **compareTo** pour comparer deux chaines de caractères. Nous pouvons utiliser cette méthode pour définir un comparateur de personnes se basant sur l'ordre lexicographique de leurs noms :

```
import java.util.Comparator;
public class TriPersonneSelonNom implements Comparator<Personne> {
	public int compare(Personne p1, Personne p2) {
		return p1.nom.compareTo(p2.nom);
	}
}
```

Les comparateurs sont aussi utiles pour les algorithmes de recherche. En effet analogiquement aux algorithmes de tri, il est parfois nécessaire d'avoir plusieurs critères de recherche pour un type de données. Par exemple, pour la classe personne il est possible d'effectuer une recherche par nom, par prénom ou par âge. Dans ces cas de figure les comparateurs sont utiles dans la mesure où ils donnent le moyen de comparer deux objets en termes d'ordre et d'égalité.

La classe Collections propose la méthode `binarySearch(collection, objetcComparateur)`. Celle-ci prend comme paramètres la collection, l'objet recherché et un comparateur par rapport au critère de recherche. Un comparateur mis à nul indique que l'ordre naturel (définit dans la clase par la méthode compareTo) devrait être utilisé.

Attention : La méthode binarySearch implémente un algorithme de recherche binaire. Ceci dit, il est impératif de trier la collection avant de l'invoquer. Dans le cas contraire, le résultat retourné est indéterminé.

La méthode binarySearch retourne :

- Un entier positif si l'élément existe.
- Si l'élément n'existe pas, la méthode retourne (- PositionDInsertion -1), où PositionDInsertion indique la position où l'élément serait positionné s'il est inséré dans la collection.

## 2.3 Le tri des tableaux

Nous avons utilisé la classe Collections pour trier les collections. Qu'en est-il pour les tableaux de type **Array** ?

En fait, le tri des tableaux de type **Array** est similaire à celui des **collections**. En effet, la classe **Arrays** propose aussi la méthode **sort**. De manière analogue à la casse **Collections**, la classe **Arrays** propose les mêmes méthodes de tri statiques et surchargées :

- `Arrays.sort(tableau)`
- `Arrays.sort(tableau, Comparator)`