

Collections en Java : Notions Intermédiaires

par Frédéric Mora ([Mes autres publications](#))

Date de publication : 07/06/2006

Dernière mise à jour : 01/02/2008

Découvrir les collections Java est une chose, en connaître les subtilités en est une autre. Cet article vous présente certaines des subtilités utiles des collections en Java.

- 1 - Introduction
- 2 - L'auto-boxing
- 3 - Les Generics
 - 3.1 - Le principe
 - 3.2 - L'utilisation de l'Auto-UnBoxing et les Generics
- 4 - Le parcours d'une collection
 - 4.1 - Les Iterators
 - 4.1.1 - Le principe
 - 4.1.2 - Les iterateurs et les Generics
 - 4.2 - Les énumérations
 - 4.3 - La nouvelle boucle for
- 5 - Notions intermédiaires et fonctions
 - 5.1 - Thread Safe ?!
 - 5.2 - Les fonctions de la classe Collections
 - 5.2.1 - Le tri de listes
 - 5.2.1.1 - sort(List)
 - 5.2.1.2 - sort(List,Comparator)
 - 5.2.2 - Minimum et maximum
 - 5.2.3 - Remplacements
 - 5.2.4 - Lecture-seule
- 6 - Remerciements

1 - Introduction

Vous avez pu découvrir, à travers l'article **Collections en Java : Prise en main**, les différentes collections présentes dans Java. Vous avez aussi pu vous familiariser à ces collections au travers d'exercices.

Vous trouverez la correction, les classes correctes et commentées ici :

Lien HTTP - Lien FTP

De plus, d'autres collections, en dehors du Framework Collections du JDK, sont présentes : les Jakarta Commons Collections.

Je ne vais pas développer dessus, Sébastien Le Ray l'a très bien fait dans **son tutoriel**. Il faut savoir qu'elles existent, je vous invite vivement à lire son article.

Tout ceci n'était qu'une découverte, et les subtilités des collections sont nombreuses : entre les Generics, l'auto-boxing ou leur utilité dans le polymorphisme, il y a de nombreux aspects à explorer. C'est parti !

2 - L'auto-boxing

L'auto-boxing n'est pas une notion spécifique des collections, mais il nous facilite la vie.

Dans la version 1.4 de java, nous étions obligés d'ajouter un `new Integer(int)`, comme dans l'exemple du tutoriel sur la prise en main des collections en java :

```
Hashtable monHashtable = new Hashtable() ;
monHashtable.put(new Integer(1), "Janvier");
monHashtable.put(new Integer(2), "Fevrier");
monHashtable.put(new Integer(3), "Mars");
monHashtable.put(new Integer(4), "Avril");
monHashtable.put(new Integer(5), "Mai");
monHashtable.put(new Integer(6), "Juin");
monHashtable.put(new Integer(7), "Juillet");
monHashtable.put(new Integer(8), "Aout");
monHashtable.put(new Integer(9), "Septembre");
monHashtable.put(new Integer(10), "Octobre");
monHashtable.put(new Integer(11), "Novembre");
monHashtable.put(new Integer(12), "Decembre");
```

Cette syntaxe devient vite très lourde lors de manipulations fréquentes de types primitifs (int, float, boolean, double, ...).

L'auto-boxing nous affranchit de cette syntaxe rébarbative et nous permet de mettre directement le type primitif :

```
Hashtable monHashtable = new Hashtable() ;
monHashtable.put(1, "Janvier");
monHashtable.put(2, "Fevrier");
monHashtable.put(3, "Mars");
monHashtable.put(4, "Avril");
monHashtable.put(5, "Mai");
monHashtable.put(6, "Juin");
monHashtable.put(7, "Juillet");
monHashtable.put(8, "Aout");
monHashtable.put(9, "Septembre");
monHashtable.put(10, "Octobre");
monHashtable.put(11, "Novembre");
monHashtable.put(12, "Decembre");
```

Bien évidemment, l'opposé est possible. Il est donc possible de récupérer un type primitif à partir de l'objet associé comme ici :

```
Integer i = 3; // Integer i = new Integer(3); -> Auto-Boxing
int j=i; // int j=i.intValue(); -> Auto-UnBoxing
```

Cette notion d'Auto-UnBoxing prend toute son utilité avec les Collections lorsqu'elle est associée à la notion de Generics.

Voici un tableau présentant l'association entre types primitifs et objets associés :

Type primitif	Objet associé
int	Integer
char	Character

float	Float
double	Double
long	Long
boolean	Boolean
byte	Byte
short	Short

3 - Les Generics

3.1 - Le principe

Lorsqu'on accède à un élément d'une collection sous Java 1.4, il est nécessaire de caster l'objet. Si l'on veut accéder à un `String` par exemple, il faut faire :

```
String s=(String)maCollection.get(i);
```

Les Generics permettent d'éviter ce cast en précisant à la création de la collection son type. Par exemple :

```
List<String> monArrayList = new ArrayList <String> (); // création d'un ArrayList qui contient des Integer
HashMap <String,Float> monHashMap = new HashMap(); // création d'un HashMap dont les clés sont des String et les valeurs sont des Float
```

Le cast disparaît. La lisibilité en est améliorée, tout comme le temps de maintenance ou la sécurité du code : un `ArrayList<Integer>` n'acceptera pas de compiler si vous essayez de lui ajouter un `String` par exemple.

3.2 - L'utilisation de l'Auto-UnBoxing et les Generics

Nous avons vu plus haut l'intérêt de l'Auto-(Un)Boxing avec les collections, notamment lors de l'ajout. Cela posait problème dans le cas de la récupération.

En effet, une collection renvoie un `Object`. Si j'essaie de stocker un `Object` (qui en fait est un `Integer`) dans un type primitif `int` comme dans cet exemple :

```
List monArrayList=new ArrayList();
monArrayList.add(1);
int n=monArrayList.get(0);
```

Une erreur est générée à la compilation : *incompatible types*, et c'est plutôt logique. L'auto-Unboxing va nous permettre de passer du type `Integer` au type `int`. Dans le cas des collections, les Generics nous servent de "casteur" pour passer de `Object` vers `Integer`. Avant, il aurait fallu faire :

```
List monArrayList=new ArrayList();
monArrayList.add(new Integer(1));
int n=((Integer)monArrayList.get(0)).intValue();
```

Maintenant, grâce à ces concepts, il suffit de faire :

```
List<Integer> monArrayList=new <Integer>ArrayList();
monArrayList.add(1); // Auto-Boxing
int n=monArrayList.get(0); // Application des Génériques (Object -> Integer) puis Auto-Unboxing (Integer -> int)
```

Là encore, la lisibilité du code est accrue, et par conséquent sa compréhension. Les casts disparaissent, les fonction superflues aussi.

4 - Le parcours d'une collection

4.1 - Les Iterators

4.1.1 - Le principe

Il est parfois bien utile de parcourir une collection sans pour autant se soucier du nombre d'éléments de cette collection. Un Iterator peut alors être utilisé pour parcourir la collections.

Cet Iterator possède une fonction *hasNext()* qui renvoie *true* s'il y a encore au moins un élément à parcourir, *false* sinon, et une fonction *next()* qui renvoie le prochaine élément à parcourir.

Pour utiliser un Iterator, il suffit d'appeler la fonction *iterator()* et de la parcourir comme ceci :

```
Iterator it=maCollection.iterator();
while (it.hasNext()) // tant que j'ai un element non parcouru
{
    Object o = it.next();
    //mes opérations
}
```

Quelques remarques :

- Un Iterator parcourt, en théorie, la liste aléatoirement.
- La fonction *next()* déplace l'itérateur, c'est à dire que faire appel trois fois à *next()* dans le même tour de boucle revient à déplacer l'itérateur de trois, ce qui peut amener des problèmes de sécurité (appel à *next()* alors qu'il n'y a plus d'éléments à parcourir).
- Les itérateurs ne peuvent pas être utilisés pour parcourir directement un Map. Il est possible cependant de le faire en parcourant les clés grâce à la fonction *iterator it = maMap.keySet().iterator()*; et en récupérant les valeurs grâce à :

```
while (it.hasNext)
{
    Objet o= maMap.get(it.next());
}
```

4.1.2 - Les itérateurs et les Generics

Il est possible d'appliquer la notion de Generics aux itérateurs. Ainsi, on peut dire à un Iterator, tout comme on peut dire à une Collection qu'elle ne contiendra qu'un type d'objet, qu'il ne va parcourir qu'un type d'objet.

Une remarque sur l'iteration : elle est *thread-safe* dans le sens que si une modification est faite en dehors de l'iterateur en cours ou par un autre thread, il en resulte une exception du type *ConcurrentModificationException* lors du prochain appel à une fonction de l'iterateur. Ce comportement est appelé *fail-fast*(échouer rapidement).

```
Iterator<String> it = maCollectionString.iterator();
while (it.hasNext()) // tant que j'ai un element String non parcouru
{
    String s = it.next();
    //mes opérations
}
```

}

4.2 - Les énumérations

Les énumérations sont similaires aux itérateurs, à cela près que les itérateurs possèdent la fonction *remove()* qui permet de retirer de la liste le dernier élément renvoyé par *next()*. De la même façon que l'on fait appel à *collection.iterator()* pour récupérer un itérateur, on fera appel à *Collections.enumeration(collection)*, ainsi que les fonctions *hasNext()* et *next()* remplacées par *hasMoreElements()* et *nextElement()*.

Pourquoi, alors, les énumérations existent-elles ? Il est possible avec ces enumerations de parcourir un HashTable, ou plutôt ses éléments / clés, ce qui ne l'était pas avec un Iterator sans passer par des fonctions comme *keySet()* et *entrySet()*.

Voici un exemple d'enumeration sur les clés d'un HashTable :

```
Enumeration enu=monHashTable.keys();
while(enu.hasMoreElements())
{
    Object o=enu.nextElement();
}
```

Il est évidemment possible d'utiliser la notion de Generics avec les Enumeration.

4.3 - La nouvelle boucle for

Java 1.5 intègre une petite nouveauté : une boucle type *for each*. A l'instar de php, java intègre donc lui aussi une boucle de parcours d'une collection.

Le principe est simple, on se rapproche du pseudo-langage : pour chaque element *e* dans la Collection *c*.

On a donc une syntaxe qui se ressemble à ça :

```
List<String> a=new ArrayList <String> ();
//remplissage de mon ArrayList avec des String
for(String current : a) // pour chaque String current dans a
{
    System.out.println(current);
}
```

Cette nouvelle boucle nous évite le problème de cast rébarbatif (tout comme les Generics), et donc améliore aussi la lisibilité.

5 - Notions intermédiaires et fonctions

5.1 - Thread Safe ?!

Après la sortie de mon premier tutoriel, beaucoup de personnes m'ont demandé pourquoi je n'avais pas développé la partie sur *thread safe*.

Comme expliqué dans ce dernier, la notion *thread safe* signifie que l'objet est pourvu de sécurités en cas d'accès simultané par deux threads. Cela signifie que des tests sont effectués à chaque accès pour savoir si un thread n'est pas déjà en train d'utiliser l'objet.

Cela provoque bien évidemment une perte de performance, pas forcément significative, mais elle existe.

Dès lors que l'on utilise le multithread, il faut prendre impérativement en compte que certains éléments peuvent être modifiés depuis un autre Thread, et que cela peut aboutir à un état incohérent.

Par exemple, la méthode *add(Object)* de *ArrayList* effectue les traitements suivants :

- Vérification de la capacité du tableau représentant la liste
- Redimensionnement du tableau si nécessaire
- Ajout de l'élément à la dernière position, et incrémentation de l'index de la dernière position

Maintenant, supposons que le tableau n'accepte plus qu'un seul élément avant d'être redimensionné, et que l'on ajoute deux éléments à partir de deux Thread différents. Voici le scénario :

- 1 Le Thread 1 vérifie la capacité de la liste, et ne redimensionne pas la liste puisqu'il reste une place libre
- 2 Le Thread 2 prend la main, vérifie la capacité de la liste, et ne redimensionne pas non plus la liste puisqu'il reste une place libre
- 3 Le Thread 2 ajoute l'élément à la dernière place et incrémente l'index de celle-ci
- 4 Le Thread 1 reprend la main et ajoute l'élément à une position incorrecte...

Une jolie exception en perspective...

Le code ci-dessous vous montre bien ce type d'erreur : il crée 20 Threads qui ajoutent en parallèle 1000 éléments dans la même liste :

```
import java.util.ArrayList;
import java.util.List;

public class Main implements Runnable
{
    private final List list;
    Main(List list)
    {
        this.list=list;
    }
    public void run()
    {
        for(int i=0; i<1000; i++)
        {
            list.add("Element "+i);
        }
        System.out.println(Thread.currentThread().getName()+ " OK...");
    }
}
```

```
}  
public static void main(String [] args)  
{  
    Main m=new Main(new ArrayList<String> ());  
    Thread[] threads=new Thread[20];  
    for(int i=0; i<threads.length; i++)  
    {  
        threads[i] = new Thread(m);  
    }  
    for(int i=0; i<threads.length; i++)  
    {  
        threads[i].start();  
    }  
}
```

Vous devriez vous retrouver avec un certain nombre d'exceptions du type : *Exception in thread "Thread-X" java.lang.ArrayIndexOutOfBoundsException*, si ce n'est pas le cas, augmentez juste le nombre de Threads et / ou d'éléments ajoutés.

Ce type d'erreur dépend en effet de l'ordre dans lequel les Threads effectuent les opérations à tour de rôle, et c'est totalement aléatoire... ce qui rend ce type d'erreur très dur à déboguer.

On dit que la méthode *add()* n'est pas *thread-safe*, puisqu'elle ne gère pas le multithreading.

Pour corriger ce problème, il faut impérativement utiliser la synchronisation avec le mot clé *synchronized* (ou la nouvelle API de concurrence du JDK 5), comme ici :

```
synchronized(reference)  
{  
    // code sécurisé  
}
```

On "verrouille" l'objet *reference*, ce qui a pour effet de garantir qu'un seul thread à la fois puisse entrer dans le bloc "sécurisé". Evidemment, tous les Thread doivent utiliser la synchronisation sur la même référence.

Ainsi, si on remplace le *run()* de l'exemple au dessus par celui ci, on n'obtient plus d'erreur :

```
public void run()  
{  
    for(int i=0;i<1000;i++)  
    {  
        synchronized(list)  
        {  
            list.add("Element "+i);  
        }  
    }  
}
```

Comme la collection utilisée est un *ArrayList*, qui n'est pas *thread-safe*, c'est à nous de gérer la synchronisation. Maintenant, si l'on remplace notre *ArrayList* par un *Vector* dans notre **premier code** (ou en créant une *List synchronized*) :

```
Main m=new Main(Collections.synchronizedList(new ArrayList<String> ()));
```

Les erreurs ont là aussi disparu.

Mais pourquoi certaines collections (Vector et HashTable) sont *thread-safe* et d'autres (ArrayList, LinkedList, HashMap, ...) pas ?

Vector et HashTable ont été hérités de Java 1.0, où la logique était de faire un maximum de choses pour assister le développeur, donc un maximum de classes *thread-safe*... Cela pose un gros problème de coût en temps d'exécution (moins significatif qu'avant).

Dans la majorité des cas, on utilisait ces classes dans un environnement mono-thread, et la synchronisation devenait inutilement coûteuse.

Java 1.2 a donc vu naître l'API de Collections et ses interfaces (Collection, List, Set, Map, SortedSet, SortedMap) et plusieurs implémentations (LinkedList, ArrayList, HashMap, ...). Sauf indication contraire, toutes ces implémentations ne sont pas *thread-safe* et les méthodes *synchronizedXXXX()* de Collections permettent d'en obtenir une version *thread-safe*.

Il est à noter que les Iterator, même après avoir synchronisé la collection, ne sont pas *thread-safe* de base, il faut gérer la synchronisation nous même.

Merci à adiGuba pour ses précisions sur ce concept.

5.2 - Les fonctions de la classe Collections

Il est très fréquent de voir des développeurs réinventer la roue, à savoir refaire des fonctions de tri, de copie, de recherche (objet, minimum, maximum), etc... Il existe une classe Collections qui permet de faire tout ça. En voici les principales fonctions.

5.2.1 - Le tri de listes

Il existe 2 fonctions de tri dans Collections : *sort(List)* et *sort(List, Comparator)*.

5.2.1.1 - sort(List)

Cette fonction trie la liste passée en paramètre en fonction de leur "ordonement naturel". Tous les éléments de la liste doivent implémenter l'interface Comparable, et donc redéfinir la fonction *compareTo(Object)*, qui renvoie 0 si les deux objets sont égaux, moins si l'objet appelant est "inférieur" et plus si l'objet appelant est "supérieur". Par exemple :

```
public class Personne implements Comparable
{
    public String nom, prenom;
    Personne(String nom, String prenom)
    {
        this.nom=nom;
        this.prenom=prenom;
    }
    public int compareTo(Object o) // on redéfinit compareTo(Object)
    {
        Personne p=(Personne)o;
        if(nom.equals(p.nom))
        {
            return prenom.compareTo(p.prenom);
        }
    }
}
```

```
    }  
    return nom.compareTo(p.nom);  
  }  
}
```

Supposons ensuite une liste *l* de *Personne*, pour la trier, il suffira d'appeler *Collections.sort(l)*.

5.2.1.2 - sort(List,Comparator)

Cette fonction trie la liste passée en paramètre en fonction du *Comparator* passé en paramètre. Supposons la classe *Personne* précédemment définie (sans implémenter *Comparable*). Il faut créer une classe implémentant l'interface *Comparator*, et donc redéfinissant les fonctions *compare(Object, Object)* et *equals(Object)*, comme cet exemple :

```
import java.util.Comparator;  
public class MonComparator implements Comparator  
{  
    MonComparator()  
    {  
    }  
    public int compare(Object arg0, Object arg1)  
    {  
        Personne p1 = (Personne) arg0;  
        Personne p2 = (Personne) arg1;  
  
        int result = p1.name.compareTo(p2.name);  
  
        if(result==0)  
            result = p1.prenom.compareTo(p2.prenom);  
  
        return result;  
    }  
}
```

Il suffit alors, pour trier ma liste *l*, d'appeler *Collections.sort(l, new MonComparator())*.

5.2.2 - Minimum et maximum

L'appel aux méthodes *Collections.min(List)*, *Collections.max(List)*, *Collections.min(List, Comparator)* et *Collections.max(List, Comparator)* requièrent les mêmes conditions d'implémentation des interfaces *Comparable* et *Comparator* des fonctions *sort(List)* et *sort(List, Comparator)*.

5.2.3 - Remplacements

Il est souvent utile de pouvoir remplacer toutes les occurrences d'un objet par un autre.

Il existe pour cela une fonction *replaceAll(List l, Object old, Object new)*. Elle prend en paramètre la liste à modifier, l'objet qui est à remplacer, et l'objet par lequel on va le remplacer.

Toutes les occurrences de cet objet telles que *old.equals(occurrence)==true* (et *old !=null*) seront remplacées par l'objet *new* passé en paramètre. Cette fonction renvoie vrai si il y avait au moins un element *e* tel que *e.equals(old)==true*, false sinon.

5.2.4 - Lecture-seule

Une fonction peu connue et pourtant bien pratique de Collections est la fonction *unmodifiableCollection(Collection)*. Elle retourne une collection identique à celle passée en paramètre à ceci près qu'elle n'est plus modifiable.

Toute tentative de modification de cette collection résulte à une exception du type *UnsupportedOperationException*. Pratique pour contrôler les faits des utilisateurs malveillants... ;).

6 - Remerciements

Je ne le remercierai jamais assez, **Pill_S** qui comme toujours est *useful* et *friendlyful* XD.

Merci à la rédaction et au staff java pour leur aide précieuse. Merci à developpez.com d'avoir accepté d'héberger mes articles.

Merci à hiko-seijuro pour sa relecture.

Et enfin, on ne leur dit peut-être pas assez, merci à tous ces gens qui sont "dans l'ombre", qui répondent aux questions sur le forum, qui donnent de leur personne pour aider les autres.

