

Collections en Java : Prise en main

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

Date de publication : 17/05/2006

Dernière mise à jour : 14/11/2007

Il existe de multiples collections en Java, mais laquelle utiliser ? Cet article vous en présente les principales pour vous aider à choisir.

- 1 - Mais c'est quoi une collection ?!
- 2 - Les Tableaux
- 3 - Les collections de type List
 - 3.1 - Les Listes Chaînées
 - 3.2 - ArrayList
- 4 - Les collections de type Map
 - 4.1 - Les HashTable
 - 4.2 - Les HashMap
- 5 - Les collections de type Set
 - - Les HashSet
- 6 - Conclusion
- 7 - Exercices
 - 7.1 - Les Tableaux
 - 7.2 - Les Listes Chaînées
 - 7.3 - Les Hashmap
- 8 - Remerciements et Sources

1 - Mais c'est quoi une collection ?!

Une collection représente un groupe d'objets, connu par ses éléments. Certaines collections acceptent les doublons, d'autres pas. Certaines sont ordonnées, d'autres pas.

Certaines collections émettent quelques restrictions, comme le type ou l'interdiction de la valeur null.

Toutes les collections en JAVA implémentent l'interface Collection par le biais de sous interfaces comme Set, Map ou List. On peut citer ArrayList, HashSet, HashMap ou LinkedList comme collections connues.

Nous aborderons dans cet article les collections du JDK 1.4, et non celles du 1.5. Nous n'aborderons donc pas les Generics, concept permettant d'éviter les casts rébarbatifs et dangereux, mais nécessaires, notamment si l'on utilise le concept de polymorphisme. Pour plus d'informations sur les Generics, je vous invite à consulter le FAQ :

http://java.developpez.com/faq/java/?page=langage_notions#NOTION_generics.

En fin de document, certains exercices vous sont proposés pour tester vos connaissances sur les collections. Libre à vous de les faire pour bien vous familiariser avec le concept.

2 - Les Tableaux

Un tableau est une liste d'éléments. Ce n'est pas une collection en soi, car un tableau n'implémente pas l'interface Collection. Cependant, son utilité étant familière, nous les abordons. La taille et le type des éléments d'un tableau sont fixés à la création de de dernier. Contrairement à une vraie collection, un tableau n'accepte pas forcément un Object, mais peut aussi accepter un type primitif comme int ou char.

Un tableau n'est pas Thread Safe à la base, c'est à dire qu'il n'a pas de protection particulière dans son implémentation si deux Threads accèdent en même temps au tableau et qu'au moins l'un d'entre eux le modifie. Voici l'exemple d'opérations sur un tableau d'entiers :

```
int [] monTableau = new int[20] ; // création d'un tableau de 20 entiers
monTableau[0]=15 ; // insertion au premier index de l'entier 15
System.out.println("Mon entier vaut : " + monTableau[0]) ; // affichage
```

Les principales fonctions des tableaux diffèrent selon le type du tableau.

Vous pouvez les retrouver sur la JavaDoc :

<http://javasearch.developpez.com/j2se/1.4.2/docs/api/java/util/Arrays.html>.

3 - Les collections de type List

Une liste est une collection ordonnée. L'utilisateur de celle-ci a un contrôle complet sur les éléments qu'il insère dedans, et il peut y accéder par l'entier de leur index.

Les listes ont toujours comme premier élément 0.

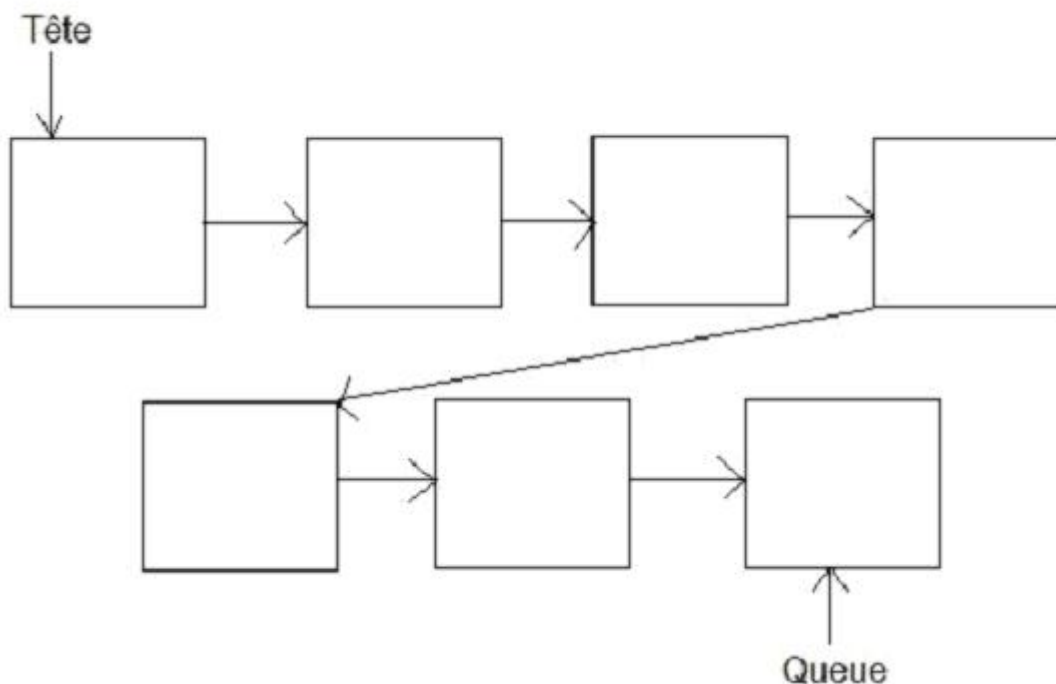
3.1 - Les Listes Chaînées

Une liste chaînée est une liste dont chaque élément est relié au suivant par une référence à ce dernier.

Une liste chaînée peut être utilisée dans le cadre d'une simulation de pile ou de file, FIFO(First In First Out) ou FILO(First In Last Out).

La taille d'une LinkedList n'est pas fixe : on peut ajouter et enlever des éléments selon nos besoins. Nous pouvons aussi remarquer que les LinkedList acceptent tous types d'objets. LinkedList n'est pas Thread Safe.

Pour bien comprendre le principe des listes chaînées, rien de mieux qu'un petit schéma :



Chaque élément contient une référence sur son suivant. On remarque que la queue n'a pas de suivant. Son suivant est en fait null.

Voici l'exemple d'opérations sur une liste chaînée :

```
List maListe=new LinkedList() ; // on crée notre liste chaînée
maListe.add(new Integer(1)) ; // on ajoute l'entier 1 à la liste
maListe.add(new Float(2.15)) ; // on ajoute le flottant 2.15 à la liste
/* On remarque ici que l'entier 1 est la tête de la liste, et que le flottant
 * est la queue de la liste. */
Integer monEntier=(Integer)maListe.getFirst() ; // on n'oublie pas de faire le cast
Float monFloat=(Float)maListe.getLast(); // on n'oublie pas de faire le cast
maListe.remove(0) ; // on retire l'entier , le flottant devient la tete
```

Soulignons que la complexité des opérations effectuées sur une `LinkedList` (liste chaînée) est proportionnelle à l'index sur lequel on effectue ces opérations. Ainsi, une opération sur l'index 0 prendra moins de temps qu'une opération sur l'index 25, ce qui est logique : lorsqu'on effectue une opération sur un élément n d'une liste chaînée, la JVM doit passer par les $n-1$ précédents de l'élément n . Il est aussi à noter que l'ajout en fin de liste est très peu coûteux en ressources machine.

Une application concrète des listes chaînées peut être une pile d'exécution. En effet, lors de l'exécution d'un programme, une pile d'exécution est créée. Lors de l'appel d'une procédure, l'adresse du code appelant est stocké dans la pile d'exécution, pour y revenir lorsque la procédure est terminée.

Les opérations principales sur une liste chaînée sont :

- `add(Object o)` : ajoute un objet `o` en fin de liste
- `addFirst(Object o)` : ajoute un objet `o` en début de liste
- `addLast(Object o)` : ajoute un objet `o` en fin de liste
- `clear()` : vide la liste
- `contains(Object o)` : renvoie `true` si la liste contient l'objet `o`, `false` sinon. La redéfinition de la fonction `hashCode()` expliquée dans la partie sur les `Map` est ici importante. Si vous utilisez la fonction `hashCode()` définie dans `Object`, `contains(Object o)` vous renverra `true` si l'objet référencé est présent dans la liste, et non pas si il y-a un objet exactement égal à `o`.
- `getFirst()` et `getLast()` : renvoie le premier et le dernier élément de la liste sous forme d'`Object`. Ne pas oublier le cast.
- `removeFirst()` et `removeLast()` : retire le premier et le dernier élément de la liste.
- `size()` : renvoie la taille de la liste

Toutes les opérations disponibles sur les `LinkedList` sont sur la Javadoc :

<http://javasearch.developpez.com/j2se/1.4.2/docs/api/java/util/LinkedList.html>.

3.2 - ArrayList

Un `ArrayList` est un tableau qui se redimensionne automatiquement. Il accepte tout type d'objets, `null` y compris. Chaque instance d'`ArrayList` a une capacité, qui définit le nombre d'éléments qu'on peut y stocker. Au fur et à mesure qu'on ajoute des éléments et qu'on " dépasse " la capacité, la taille augmente en conséquence.

`ArrayList` n'est pas `Thread Safe`.

Voici un exemple d'opérations sur un `ArrayList` :

```
List monArrayList = new ArrayList(10) ; // on crée un ArrayList de taille 10
for(int i=0;i<30;i++) {
    monArrayList.add(new Integer(i)); // on ajoute 30 entiers
}
/* On remarque que le ArrayList , au moment d'ajouter
 *10, atteint sa capacité maximale, il se redimensionne pour en
 * accepter plus */
```

Les opérations d'accès à un objet (get), de récupération de la taille (size), d'itération (iterator) et d'itération spéciale (listIterator) sont exécutées en temps constant, c'est à dire que quel que soit le ArrayList, ces opérations prendront le même temps.

L'opération d'ajout quant à elle est en temps amorti, c'est à dire que les ajouts en fin et début de liste sont plus rapides. Cependant, il faut mettre en avant la faiblesse d'ArrayList qui est que le fait de redimensionner demande beaucoup de ressources. Il est donc nécessaire de minimiser ces redimensionnements, notamment en définissant une taille de base assez large lors de sa construction.

Les opérations principales sur un ArrayList sont :

- add(Object o) : ajoute l'objet o à la fin du ArrayList
- clear() : vide le ArrayList
- get(int index) : renvoie l'Object à l'index spécifié. Renvoie une exception si vous dépassez le tableau (IndexOutOfBoundsException)
- size() : renvoie la taille du ArrayList

Toutes les opérations disponibles sur les ArrayList sont disponibles sur la Javadoc :

<http://javasearch.developpez.com/j2se/1.4.2/docs/api/java/util/ArrayList.html>

4 - Les collections de type Map

Une map est une collection qui associe une clé à une valeur. La clé est unique, contrairement à la valeur qui peut être associée à plusieurs clés. La majorité des collections de type Map ont deux constructeurs : un constructeur sans paramètre créant une Map vide, et un constructeur prenant en paramètre une Map qui crée une nouvelle Map en fonction de la Map passée en paramètre.

4.1 - Les HashTable

Un HashTable est une implémentation de Map qui associe une clé à une valeur. N'importe quel objet, mis à part null peut y être ajouté.

Voici un exemple d'utilisation de HashTable, où on associe un numéro de mois à son nom :

```
Map 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), "Décembre");
```

La complexité d'accès à une valeur par sa clé dépend d'une fonction appelée hashCode définie dans la clé. Cette fonction est déjà définie dans Object, mais pour optimiser la recherche, la surcharger dans vos propres classes (si la clé est une instance d'une classe de votre création) permet un gain de temps non négligeable. Le principe de cette fonction est simple : elle renvoie un entier. Si les deux objets sont égaux, les valeurs renvoyées par leur fonction respectives sont égales, mais l'inverse n'est pas vrai, ce qui veut dire que si deux objets sont inégaux, leurs hashCode ne sont pas forcément inégaux.

Généralement, on fait renvoyer à la fonction hashCode() d'un objet la somme des hashCode() des éléments le composant. Le HashTable est Thread Safe.

Les opérations disponibles sur un HashTable sont sur la Javadoc :

<http://javasearch.developpez.com/j2se/1.4.2/docs/api/java/util/Hashtable.html>

4.2 - Les HashMap

Un HashMap est une collection similaire au HashTable. Seules deux choses les diffèrent : HashMap accepte comme clé et comme valeur null, et HashMap n'est pas Thread Safe.

Toutes les opérations disponibles sur le HashMap sont sur la JavaDoc :

<http://javasearch.developpez.com/j2se/1.4.2/docs/api/java/util/HashMap.html>

5 - Les collections de type Set

Un Set est une collection qui n'accepte pas les doublons, et au maximum une fois null. Cela veut dire que deux éléments e1 et e2 d'un Set ne seront pas égaux par e1.equals(e2).

Certains Set sont plus restrictifs que d'autres, n'acceptent pas null ou un certain type d'objet. Dans le cas où un ajout d'un élément déjà présent dans le Set venait à se faire, une exception de type NullPointerException ou ClassCastException serait levée et traitée.

- - Les HashSet

HashSet est l'implémentation la plus utile de Set. Elle permet de stocker des objets sans doublons. Comme dit plus haut, des exceptions sont levées si l'on essaie d'insérer un objet non autorisé (comme *null*). Cependant, si l'on essaie d'ajouter un objet déjà présent, la fonction *add* renverra tout simplement *false* et ne fera rien. Pour parcourir un HashSet, on est obligé d'utiliser un Iterator.

Il est à signaler que les Map expliquées plus haut utilisent les Set pour garder les clés uniques.

Voici un exemple d'opérations sur un HashSet :

```
Set monHashSet=new HashSet(); // on crée notre Set
monHashSet.add(new String("1")); // on ajoute des string quelconques
monHashSet.add(new String("2"));
monHashSet.add(new String("3"));
monHashSet.add(new String("1")); // oups, je l'ai déjà ajouté, la fonction gère l'exception levée,
    et l'objet n'est pas ajouté
Iterator i=monHashSet.iterator(); // on crée un Iterator pour parcourir notre HashSet
while(i.hasNext()) // tant qu'on a un suivant
{
    System.out.println(i.next()); // on affiche le suivant
}
/* Notez que l'itération se fait aléatoirement. */
```

6 - Conclusion

Java contient un grand nombre de collections différentes, toutes adaptées à des situations différentes. Le tout est de bien choisir la collection en fonction de ses besoins.

Cet article n'est qu'une introduction aux collections, nous n'avons pas abordé des points comme les Generics disponibles dans le JDK 1.5, mais nous avons vu les bases nécessaires à l'utilisation de ces collections en Java.

Et maintenant, place à la pratique (c'est ça le plus amusant ;).

7 - Exercices

Ces exercices sont là dans un but ludique.

Ils ne sont pas obligatoires, mais vont vous permettre de manipuler certaines structures pour mieux en comprendre le fonctionnement.

La correction de ces exercices sera donnée dans une autre publication, le temps de vous laisser chercher un peu.

7.1 - Les Tableaux

Créer une classe `Tableau` qui a comme attribut un tableau d'entiers. Le constructeur par défaut initialise ce tableau à 10 éléments. Elle possède également une fonction qui remplit le tableau de valeurs au choix, et une fonction qui affiche les valeurs contenues dans le tableau. Créer une fonction `sort()` qui trie le tableau. Vous pouvez utiliser une fonction de tri déjà définie dans le JDK.

Créer une classe `AppliTableau` qui crée un objet `Tableau`, qui le remplit, l'affiche, le trie et le réaffiche.

7.2 - Les Listes Chaînées

Le but de cet exercice est de vous faire manipuler les listes chaînées par un exemple concret : une file d'attente.

Supposons une file d'attente de cinéma. Sa capacité de stockage est illimitée (c'est un cinéma géant). Une personne est définie par son nom et son prénom.

- 1 Créer une classe `Personne` contenant un champ pour le nom et un champ pour le prénom, un constructeur prenant en paramètre deux `String` pour le nom et le prénom, et une fonction `toString` retournant le nom et le prénom sous forme de `String`.
- 2 Compléter cette classe pour ajouter une référence sur la personne suivante dans la file (celle qui attend derrière). Ajouter une fonction `setSuivant(Personne p)` qui définit le suivant de la personne et une fonction `getSuivant()` qui retourne la `Personne` suivante dans la file.
- 3 Créer une classe `FilleAttente`. Cette classe possède deux champs de type `Personne` : un champ `tete` (début de la file d'attente) et un champ `queue` (fin de la file d'attente). Le constructeur prend en paramètre 1 objet de type `Personne` et remplit `tete` et `queue` (la première personne est la tête et la queue en même temps). Créer une fonction `ajouter(Personne p)` qui ajoute une personne en bout de file. Créer la fonction `afficherFile()` qui affiche les personnes qui attendent dans la file d'attente. Un exemple d'affichage de cette fonction pourrait être :

Jojo Lafrousse - Pierre Delacroix - Jean Lartige - Fin de la file !

- 4 Créer une classe `AppliFile`. Cette classe crée une file d'attente contenant une personne, y ajoute 5 personnes (faut bien faire bosser le caissier) avec les noms de votre choix, et l'affiche.

7.3 - Les Hashmap

Une application des `Map` peut être l'optimisation, par exemple, d'interfaces graphiques. Supposons une application graphique, un jeu. Ce jeu affiche un joueur à l'écran.

On dispose d'une classe dans laquelle il y a une fonction `chargerImage(String s)` qui retourne une instance d'objet `BufferedImage`. On suppose que le `String` passé en paramètre est toujours une image chargeable.

En voici le code :

```
import java.awt.image.BufferedImage;
import java.net.URL;
import javax.imageio.ImageIO;
public class AppliMap
{
    AppliMap()
    {
    }
    public BufferedImage chargerImage(String path) // ma fonction
    {
        URL url; // notre url permettant de charger
        try
        {
            url=getClass().getClassLoader().getResource(path); // on charge l'URL par rapport au path
            return ImageIO.read(url); // et on la retourne
        } catch (Exception e) // si ca s'est mal passé (et là c'est le drâme ! ^^)
        {
            System.out.println("Impossible de charger le fichier : "+path);
            System.out.println("L'erreur retournée : "+e.getMessage());
            System.exit(0);
            return null;
        }
    }
}
```

En l'état actuel des choses, ce code est très lent dans l'optique de chargement d'images fréquent. En effet, lors de l'appel à cette procédure, une nouvelle instance de `BufferedImage` est créée. Si l'on charge plusieurs fois la même image, on crée autant d'instance de `BufferedImage`. Pas glop me dirrez vous.

Le but de cet exercice est de modifier cette classe, en utilisant un `HashMap` ou un `HashTable`, pour qu'on ne crée qu'une fois par fichier l'instance d'une image. A vous de jouer ;).

8 - Remerciements et Sources

Je tiens à remercier **Pill_S** pour sa disponibilité, sa gentillesse et son aide précieuse.

Je remercie également tous les modérateurs et rédacteurs JAVA pour leurs réactions et leurs aides par l'intermédiaire de Pill_S.

Je remercie enfin mes profs de DUT, notamment M. Audemard, pour ses cours sur JAVA, qui m'ont bien servi mine de rien ;).

Sources :

FAQ Java : <http://java.developpez.com/faq/>

JavaSearch : <http://javasearch.developpez.com/>

Cours de Java, M. Audemard, IUT de Lens

