Collections en Java[1]

[1] Bibliographie:

Ce document est inspiré grandement de la page web de SUN sur les collections, qui se trouve à:

http://developer.java.sun.com/developer/onlineTraining/collections/Collection.html

et les notes de cours du Prof. Guy Lapalme.

1. Structures de données

C'est l'organisation efficace d'un ensemble de données, sous la forme de tableaux, de listes, de piles etc. Cette efficacité réside dans la quantité mémoire utilisée pour stocker les données, et le temps nécessaire pour réaliser des opérations sur ces données.

2. Collections & Java

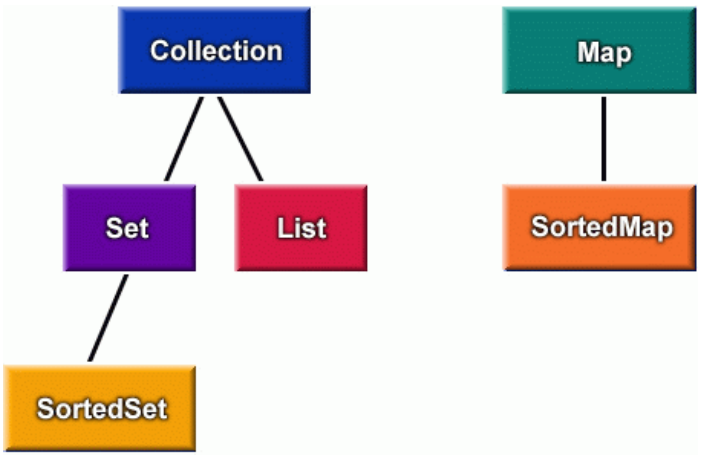
Une collection gère un groupe d'un ensemble d'objets d'un type donné ; ou bien c'est un objet qui sert à stocker d'autres objets. Dans les premières versions de Java, les collections étaient représentées par les "Array","Vector","Stack" etc. Puis avec Java 1.2 (Java 2), est apparu le framWork de collections qui tout en gardant les principes de bases, il a apporté des modifications dans la manière avec laquelle ces collections ont été réalisées et hiérarchisées.

Tout en collaborant entre elles, ces collections permettent de réaliser dans des catégories de logiciels des conceptions réutilisables.

3. Collections Framwork de Java

Réparties en deux groupes:

3.1. Interfaces



Organisées en deux catégories: Collection & Map.

- Collection: un groupe d'objets où la duplication peut-être autorisée.

- Set: est ensemble ne contenant que des valeurs et ces valeurs ne sont pas dupliquées. Par exemple l'ensemble A = {1,2,4,8}. Set hérite donc de **Collection**, mais n'autorise pas la duplication. SortedSet est un Set trié.

- List: hérite aussi de collection, mais autorise la duplication. Dans cette interface, un système d'indexation a été introduit pour permettre l'accès (rapide) aux éléments de la liste.

- Map: est un groupe de paires contenant une clé et une valeur associée à cette clé. Cette interface n'hérite ni de Set ni de Collection. La raison est que Collection traite des données simples alors que Map des données composées (clé,valeur). SortedMap est un Map trié.

3.2. Implémentations

Le framework fournit les implémentations suivantes des différentes interfaces:

	Classes d'implémentations				
		Table de Hachage	Tableau de taille variable	Arbre balancé	Liste chaînée
Interfaces	Set	HashSet		TreeSet	
	List		ArrayList		LinkedList
	Map	HashMap		TreeMap	

Par contre, il n'y a pas d'implémentation de l'interface Collection. Pour Set et Map l'implémentation est soit sous la forme d'une table de hachage (HashSet/HashMap) ou bien sous la forme d'un arbre (TreeSet/TreeMap). Pour la liste: soit sous la forme de tableau (ArrayList) ou une liste chaînée (LinkedList).

4. Algorithmes

Sont utilisés pour traiter les éléments d'un ensemble de données. Ils définissent une procédure informatique, par exemple: tris, recherche etc.

5. Itérateurs

Fournissent aux algorithmes un moyen pour parcourir une collection du début à la fin. Ce moyen permet de retirer donc à la demande des éléments donnés de la collection.

6. Description des interfaces

6.1. Collection

```
public interface Collection {

        // Basic Operations
        int size();
        boolean isEmpty();
        boolean contains(Object element);
        boolean add(Object element); // Optional
        boolean remove(Object element); // Optional
        Iterator iterator();


        int hashCode();
        boolean equals(Object element);

        // Bulk Operations
        boolean containsAll(Collection c);
        boolean addAll(Collection c); // Optional
        boolean removeAll(Collection c); // Optional
        boolean retainAll(Collection c); // Optional
        void clear(); // Optional

        // Array Operations
        Object[] toArray();
        Object[] toArray(Object a[]);
}
```

Les interfaces contiennent des méthodes optionnelles. Cette approche permet de traiter les collections particulières sans que nous soyons dans l'obligation de définir les méthodes optionnelles. Ces méthodes optionnelles sont définies qu'en cas de besoin. Un Set non modifiable n'a pas besoin de redéfinir la méthode add, puisque nous ne pouvons pas le modifier!

Il y a des opérations réalisées sur un seul objet ou bien sur une collection (un ensemble d'objets).

add (remove) permet d'ajouter (resp. de retirer) un élément. Quand à addAll (removeAll) permet d'ajouter (resp. de retirer même si les éléments sont dupliqués dans la collection originale) une collection.

contains (containsAll) permet de vérifier si un objet (resp. les éléments d'une collection) est présent dans la collection.

size, isEmpty et clear, permettent respectivement de donner la taille de la collection, de vérifier si la collection est vide et finalement d'effacer le contenu de la collection.

retainsAll se comporte comme le résultat de l'intersection de deux ensembles. Si A={1,2,5,8} et B={3,8} alors A = {8}.

equals permet de tester si deux objets sont égaux.

hashCode retourne le code de hachage calculé pour la collection.

toArray retourne les éléments de la collection sous le format d'un tableau.

toArray(Object a[]) permet de préciser le type du tableau à retourner. Si le tableau est grand les éléments sont rangés dans ce tableau, sinon un nouveau tableau est crée pour recevoir les éléments de la collection.

L'interface collection est dotée d'une instance d'une classe qui implante l'interface Iterator. C'est l'outil utilisé pour parcourir une collection. L'interface Iterator contient ce qui suit:

```
public interface Iterator {
       boolean hasNext();
       Object next();
       void remove(); // Optional
}
```

hasNext permet de vérifier s'il y a un élément qui suit.
next permet de pointer l'élément suivant.
remove permet de retirer l'élément courant.

```
Collection collection = ...;
Iterator iterator = collection.iterator();
while (iterator.hasNext()) {
      Object element = iterator.next();
      if (removalCheck(element)) {
       iterator.remove();
      }
}
```

Les collections vues comme des ensembles réalisent les 3 opérations mathématiques sur des ensembles:

union: add et addAll

intersection: retainAll

différence: remove et removeAll

6.2. Set

C'est une interface identique à celle de Collection. Deux implémentations possibles:

TreeSet: les éléments sont rangés de manière ascendante.
HashSet: les éléments sont rangés suivant une méthode de hachage.

```
import java.util.*;

public class SetExample {
       public static void main(String args[]) {
            Set set = new HashSet(); // Une table de Hachage
            set.add("Bernadine");
              set.add("Elizabeth");
              set.add("Gene");
              set.add("Elizabeth");
              set.add("Clara");
              System.out.println(set);
              Set SetTrie = new TreeSet(set); // Un Set trié
              System.out.println(sortedSet);
       }
}
[Gene, Clara, Bernadine, Elizabeth]
[Bernadine, Clara, Elizabeth, Gene]
```

6.2. List

Liste est une collection ordonnée. Elle permet la duplication des éléments. L'interface est renforcée par des méthodes permettant d'ajouter ou de retirer des éléments se trouvant à une position donnée. Elle permet aussi de travailler sur des sous listes. On utilise le plus souvent des ArrayList sauf s'il y a insertion d'élément(s) au milieu de la liste. Dans ce cas il est préférable d'utiliser une LinkedList pour éviter ainsi les décalages.

```
public interface List extends Collection {

       // Positional Access
       Object get(int index);
       Object set(int index, Object element); // Optional
       void add(int index, Object element); // Optional
       Object remove(int index); // Optional
       boolean addAll(int index, Collection c); // Optional

       // Search
       int indexOf(Object o);
       int lastIndexOf(Object o);

       // Iteration
       ListIterator listIterator();
       ListIterator listIterator(int index);

       // Range-view
       List subList(int fromIndex, int toIndex);
}
```

Les méthodes de l'interface List permettent d'agir sur un élément se trouvant à un index donné ou bien un ensemble d'éléments à partir d'un index donné dans la liste.

get (remove) retourne (resp. retirer) l'élément se trouvant à la position index.
set (add & addAll) modifie (resp. ajouter) l'élément (resp. un seul ou une collection) se trouvant à la position index.

indexOf (lastIndexOf) recherche si un objet se trouve dans la liste et retourner son (resp. son dernier) index.

subList permet de créer un sous liste d'une liste.

Pour parcourir une liste, il a été défini un itérateur spécialement pour la liste.

```
public interface ListIterator extends Iterator {

        boolean hasNext();
        Object next();

        boolean hasPrevious();
        Object previous();

        int nextIndex();
        int previousIndex();

        void remove(); // Optional
        void set(Object o); // Optional
        void add(Object o); // Optional
}
```

permet donc de parcourir la liste dans les deux directions et de modifier un élément (set) ou d'ajouter un nouveau élément.

```
List list = ...;
ListIterator iterator = list.listIterator(list.size());
while (iterator.hasPrevious()) {
      Object element = iterator.previous();
      // traitement d'un élément
}
```

hasNext permet de vérifier s'il y a un élément qui suit.
next permet de pointer l'élément courant.
nextIndex retourne l'index de l'élément courant.

Pour les sous listes, elles sont extraites des listes de fromIndex (inclus) à toIndex (non inclus). Tout changement sur les sous listes affecte la liste de base, et l'inverse provoque un état indéfini s'il y a accès à la sous liste.

```
import java.util.*;

public class ListExample {
       public static void main(String args[]) {
            List list = new ArrayList();
            list.add("Bernadine");
            list.add("Elizabeth");
            list.add("Gene");
            list.add("Elizabeth");
            list.add("Clara");
            System.out.println(list);
            System.out.println("2: " + list.get(2));
            System.out.println("0: " + list.get(0));
            LinkedList queue = new LinkedList();
            queue.addFirst("Bernadine");
            queue.addFirst("Elizabeth");
            queue.addFirst("Gene");
            queue.addFirst("Elizabeth");
            queue.addFirst("Clara");
            System.out.println(queue);
            queue.removeLast();
            queue.removeLast();
            System.out.println(queue);
       }
}
```

```
Bernadine, Elizabeth, Gene, Elizabeth, Clara]
2: Gene
0: Bernadine
[Clara, Elizabeth, Gene, Elizabeth, Bernadine]
[Clara, Elizabeth, Gene]
```

6.3. Map

C'est un ensemble de paires, contenant une clé et une valeur (en réalité, nous pouvons associer plusieurs valeurs. Dans ce cas la, nous sommes en présence d'une multimap … à voir en démo.).

Deux clés ne peuvent être égales au sens de equals.

L'interface interne Entry permet de manipuler les éléments d'une paire comme suit:

```
public interface Entry {
        Object getKey();
        Object getValue();
        Object setValue(Object value);
    }
```

getKey & getValue retournent respectivement la clé et la valeur associée à cette clé. setValue permet de modifier une valeur d'une paire. Remarque: faire attention de ne pas modifier directement la valeur associée à une clé. Pour le faire, retirer l'ancienne clé (et donc sa valeur aussi) et ajouter une nouvelle clé (avec cette nouvelle valeur).

```
public interface Map {

    // Basic Operations
    Object put(Object key, Object value);
    Object get(Object key);
    Object remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();

    // Bulk Operations
    void putAll(Map t);
    void clear();

    // Collection Views
    public Set keySet();
    public Collection values();
    public Set entrySet();

    // Interface for entrySet elements
    public interface Entry {
        Object getKey();
        Object getValue();
        Object setValue(Object value);
    }
}
```

values retourne les valeurs sous la forme d'une Collection.
keySet et entrySet retournent, respectivement, un ensemble (Set) de clés et un ensemble (set) d'Entry.

Ceci permet donc d'itérer sur les Map comme suit:

si m est un HashMap alors:

```
// sur les clés
for (Iterator i = m.keySet().iterator();i.hasNext();)
      System.out.println(i.next());

// sur les valeurs
for (Iterator i = m.values().iterator();i.hasNext();)
      System.out.println(i.next());

// sur la paire clé/valeur
for (Iterator i = m.keySet().iterator();i.hasNext();){
      Map.Entry e = (Map.Entry) i.next();
      System.out.println(e.getKey() + " ; " + e.getValue());
}
```

```
import java.util.*;

public class MapExample {
    public static void main(String args[]) {
            Map map = new HashMap();
            Integer ONE = new Integer(1);
            for (int i=0, n=args.length; i<n; i++) {
                  String key = args[i];
                  Integer frequency = (Integer)map.get(key);
                  if (frequency == null) {
                        frequency = ONE;
                  } else {
                        int value = frequency.intValue();
                        frequency = new Integer(value + 1);
                  }
                  map.put(key, frequency);
            }
            System.out.println(map);
            Map sortedMap = new TreeMap(map);
            System.out.println(sortedMap);
    }
}
```

7. Description des algorithmes

L'ensemble des algorithmes manipulant les collections se trouve dans la classe Collections (à ne pas confondre avec l'interface Collection). Ces méthodes ont été définies statiques.

Des algorithmes qui ne s'appliquent que sur des listes:

Trier:

sort(List list) ; trie une liste.
sort(List list,Comparator comp) ; trie une liste en utilisant un comparateur.

Mélanger:

shuffle(List liste) ; mélange les éléments de manière aléatoire.

Manipuler:

reverse(List liste) ; inverse les éléments de la liste.
fill (List liste, Object element) ; initialise les éléments de la liste avec element.
copy(List dest, List src) ; copy une liste src dans une liste dest.

Rechercher:

binarySearch(List list, Object element) ; une recherche binaire d'un élément.
binarySearch(List list, Object element, Comparator comp) ; une recherche binaire d'un élément en utilisant un comparateur.

Des algorithmes qui s'appliquent sur toutes les collections:

effectuer des recherches extrêmes:
min, max etc.

min (Collection)
max (Collection)

8. Interface Comparable

Un algorithme extrêmement rapide et stable (les éléments équivalents ne sont pas réordonnés) est utilisé pour trier la liste en utilisant l'ordre naturel du type. Le tri ne peut avoir lieu que si les classes implantent la méthode Comparable, ce qui n'est toujours pas le cas[2]. Cette classe contient une seule méthode compareTo:

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

Cette méthode retourne:

- entier positif si l'objet qui fait l'appel est plus grand que obj,
- zéro s'ils sont identiques,
- négatif si l'objet qui fait l'appel est plus petit que obj.

Dans le cas d'une classe qui n'implante pas la classe Comparable, ou bien vous voulez spécifier un autre ordre, vous devez implanter l'interface Comparator. Cette dernière permet de comparer deux éléments de la collection. Pour trier, nous passons une instance de cette classe à la méthode Sort().

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


class CaseInsensitiveComparator implements Comparator {
       public int compare(Object element1,Object element2) {
             String lowerE1 = (
                   (String)element1).toLowerCase();
                   String lowerE2 = (
                   (String)element2).toLowerCase();
             return lowerE1.compareTo(lowerE2);
       }
}
```

[2] C'est le cas pour les types primitifs: int, char, double, float, String etc.

9. Les librairies de collections

Pour le langage C++ il existe une bibliothèque appelée la STL (Standard Templates Library), l'équivalent pour le langage java est la JGL (Generic collections library for Java) développée par ObjectSpace. Vous pouvez la trouver à cette adresse:

http://www.objectspace.com/products/voyager/downloads.asp

Si vous décidez de la télécharger, pas besoin de remplir tous les champs. D'après Zukowski[3] (page 312), les informations fournies pour le téléchargement risqueraient de se retrouver dans la nature …

L'utilisation de la JGL n'est pas conseillée vu sa complexité. Il est conseillé d'installer la documentation associée. Il est à signaler que la JGL n'a pas été mise à jour depuis la version beta de java 1.2 ; de ce fait, la version 1.2 de java n'est pas supportée.

[3] Zukowski, John : "Java Collections". Éditeur: Apress. www.apress.com ; en réserve à la bibliothèque de math/info. Le livre contient une description des collections en java. Attention le livre contient peu d'exemples, c'est plutôt une description de point de vue technique des collections.