Programmation Orientée Objet en JAVA

Les collections

Objectifs

- ✓ Manipuler les Collections :
  - List
  - Set
  - Map
  - Queue
- ✓ Utiliser la classe « Collections »
- ✓ Utilitaires :
  - trier une collection
  - parcourir une collection
  - chercher une information dans une liste triée

Collections: présentation

- Les collections proposent une série de classes, d'interfaces et d'implémentations pour gérer efficacement les données.
- Les classes et les interfaces se trouvent dans le paquetage : java.util.

Collections: avantages

- Les collections permettent de :
  - améliorer la qualité et la performance des applications
  - gérer un groupe d'un ensemble d'objets de types différents
- Les collections sont utilisées pour:
  - stocker, rechercher et manipuler des données
  - transmettre des données d'une méthode à une autre
- Exemples :
  - un dossier de courrier : collection de mails
  - un répertoire téléphonique : collection d'associations noms/ numéros de téléphone.

Collections: architecture

- Composée de 3 parties :

- Une hiérarchie d'interfaces permettant de représenter les collections sous forme de types abstraits.

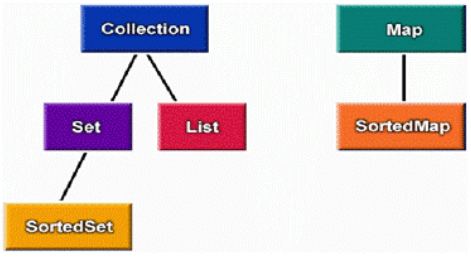
- Des implémentations de ces interfaces.

- Implémentation de méthodes liées aux collections (recherche, tri, etc.).

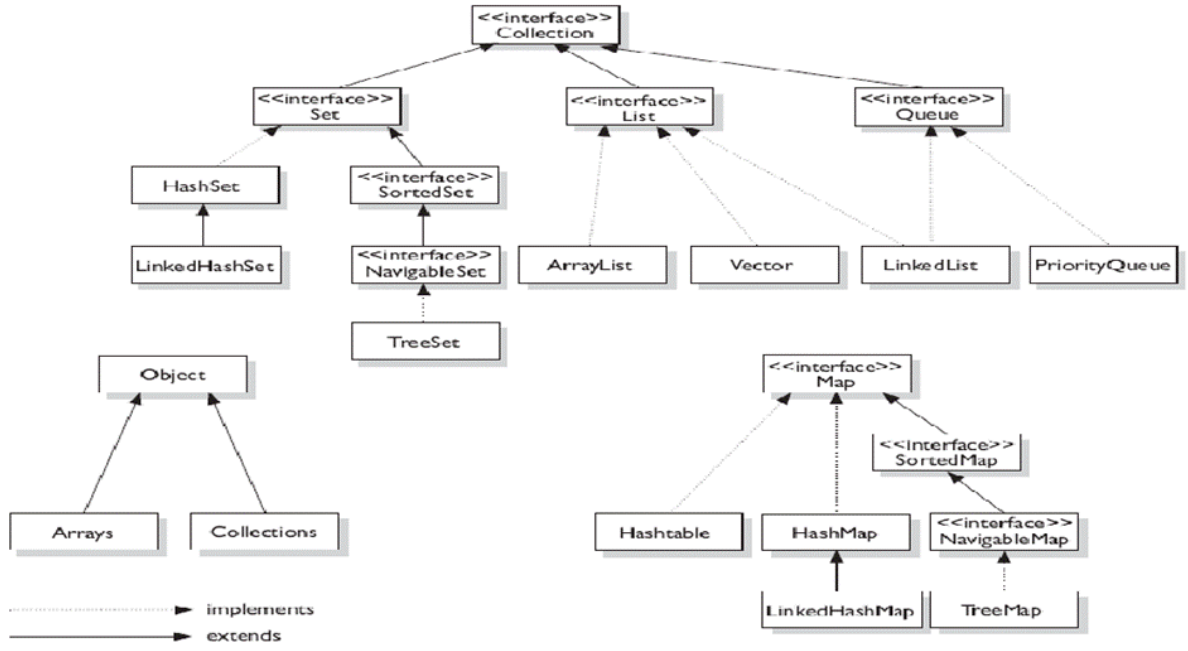
Collections: architecture

- 2 hiérarchies principales :

  - Collection
  - Map



Collections: architecture



Collections: architecture

- Les java.util.List (listes) sont une suite d'éléments ordonnés accessibles par leurs indices (leurs places dans la liste).
Les listes ne garantissent pas l'unicité des éléments.

- Les java.util.Set (ensembles) sont un groupe d'éléments uniques.

- Les java.util.Map (associations) mémorisent une collection de couples clé-valeur. Si vous avez une clé, l'association retrouvera la valeur associée à cette clé. Les clés sont uniques, mais la même valeur peut-être associée à plusieurs clés.

Interface Collections

- **C'est un objet qui est lui aussi un groupe d'objets.**
- **Principales méthodes :**

```
boolean add(Object obj)
boolean contains(Object obj)
boolean containsAll(Collection collection)
int size()
Object[] toArray()
Object[] toArray(Object[] tableau)
```


List

List

- Les objets appartenant à la catégorie List sont des tableaux extensibles à volonté.
  On y trouve les objets « Vector », « LinkedList » et « ArrayList ».
- Vous pouvez y insérer autant d'éléments que vous le souhaitez sans craindre de dépasser la taille de votre tableau.

Classe ArrayList<E> Vs vector

java.util.ArrayList

- utilise un tableau en interne pour ranger les données,
- fournit un accès aux éléments par leur indice très performant et est optimisé pour des opérations d'ajout/suppression d'éléments en fin de liste,
- Les emplacements sont repérés par des nombres entiers (à partir de 0)

java.util.Vector

- Synchronisé par défaut (4x plus lent que ArrayList)
- est une classe dite "thread-safe", c'est-à-dire que plusieurs processus peuvent l'utiliser en même temps sans risque de perte de données.

ArrayList

- La classe « ArrayList » implémente un tableau d'objets qui peut grandir ou rétrécir à la demande, ce qui débarrasse le programmeur de la gestion de la taille du tableau.
- Comme pour un tableau, on peut accéder à un élément du « ArrayList » par un indice.

ArrayList (exemple)

```
import java.util.ArrayList;
    public class Test {
    public static void main(String[] args) {

          ArrayList al = new ArrayList();
          al.add(12);
          al.add("Une chaîne de caractères !");
          al.add(12.20f);
          al.add('d');

    for(int i = 0; i < al.size(); i++) {
     System.out.println("donnée à l'indice " + i + " = " + al.get(i));
    }
}
}
```

Tri et Recherche dans List

Classe Collection

- Cette classe ne contient que des méthodes static, utilitaires pour travailler avec des collections et effectuer des opérations de :
  - tri (sur listes)
  - recherche (sur listes)
  - copie
  - recherche du minimum et du maximum

Tri

- Afin de trier une liste:

  Collections.sort(l);

- Pour que la liste soit correctement triée, il faut que les éléments de la liste soient mutuellement comparables

- Plus exactement, la méthode **sort()** ne fonctionnera que si tous les éléments de la liste doivent appartenir à une classe qui implémente l'interface **Comparable**

Tri

- Afin de trier une liste:

  Collections.sort(l);

- Pour que la liste soit correctement triée, il faut que les éléments de la liste soient mutuellement comparables

- Plus exactement, la méthode **sort()** ne fonctionnera que si tous les éléments de la liste doivent appartenir à une classe qui implémente l'interface **Comparable**

Tri

- Il est possible d'utiliser la méthode sort de la classe Collections, dans le cas ou l'on souhaite préciser le critère de tri à adopter, comme suit :

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


Interface Comparable<T>

- Cette interface correspond à l'implantation d'un ordre naturel dans les instances d'une classe.
- Redéfinir la méthode

public int compareTo (Object o) avec:

- a.compareTo (b) == 0 si a.equals To(b)
- a.compareTo (b) < 0 si a plus « petit »que b
- a.compareTo (b) > 0 si a plus « grand » que b

Interface Comparator<T>

- C'est possible de trier la liste d'une autre manière, en créant une classe qui implantera l'interface java.util.Comparator, afin de comparer deux éléments de la collection
- Il faut implémenter cette méthode:

int compare(T o1, T o2) qui doit renvoyer

- un entier positif si **o1 est « plus grand » que o2**
- **0** si **o1 a la même valeur (au sens de equals)** que **o2**
- un entier négatif si **o1 est « plus petit » que o2**

- Pour utiliser le critère de tri qui est implémenté précédemment, il faut appeler une instance de cette classe en paramètre de la méthode sort()