Les Collections

Les Collections sont des objets qui permettent de manipuler des ensembles d'objets, ou encore de manipuler les *structures de données* telles que, les vecteurs dynamiques, les ensembles, les listes chaînées, les tables associatives.

Cet ensemble de bibliothèques du paquetage java.util a été introduit à la version 2 de Java pour amener un lot de performances sur notamment la simplicité, la concision, l'universalité, l'homogénéité et la flexibilité.
Ainsi les classes recouvrant les *vecteurs*, les *listes* et les *ensembles* implémentent une même interface: l'interface Collection.
Dans ce chapitre, nous commencerons par explorer l'univers des Collections et nous aborderons l'étude des algorithmes qui nous permettront de réaliser un certain nombre d'opérations sur les Collections, à savoir: la recherche de minimum, de maximum, le tri, la recherche binaire,…
Nous terminerons sur les tables associatives qui implémentent l'interface Map.

I Présentation du framework collection

Les interfaces à utiliser par des objets qui gèrent les collections sont:

Collection : interface implémentée par la plupart des objets qui gèrent des collections
Map : interface qui définit des méthodes pour des objets qui gèrent des tables associatives sous forme clé/valeur.
Set : interface pour des objets *qui n'autorisent pas de gestion des doublons* dans l'ensemble
List : interface pour des objets *qui autorisent la gestion des doublons* et un accès direct à un élément.
SortedSet : interface qui étend l'interface Set et permet d'ordonner l'ensemble.
SortedMap : interface qui étend l'interface Map et permet d'ordonner l'ensemble.
NB: les interfaces List et Set étendent l'interface Collection, et SortedSet étend Set.

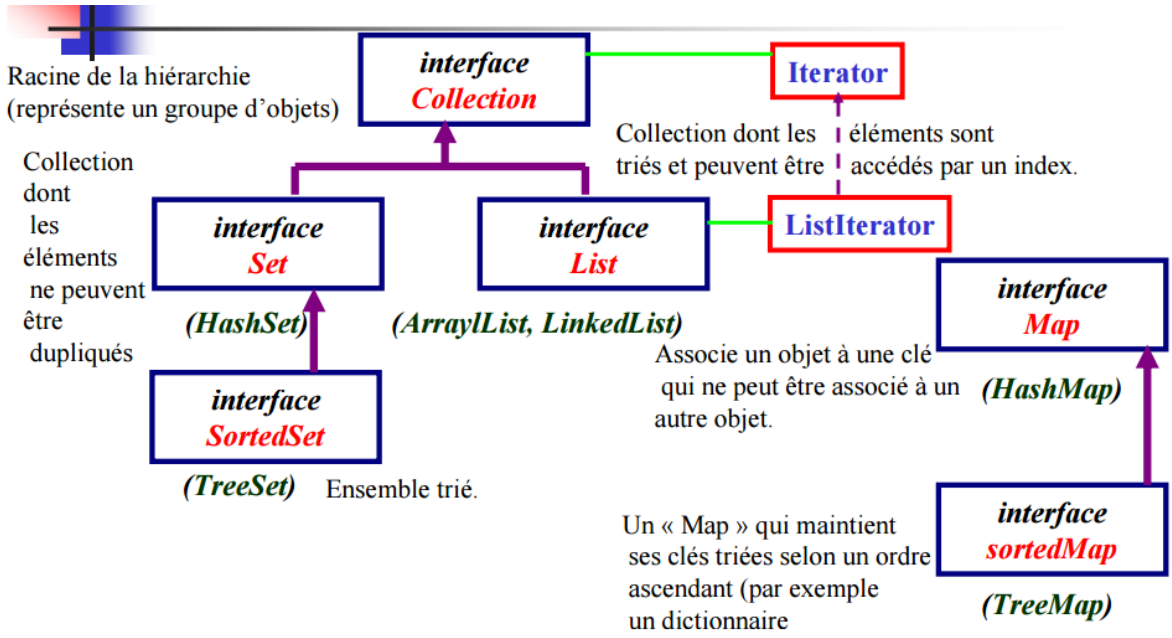
Le framework propose plusieurs objets qui implémentent ces interfaces et qui peuvent être directement utilisés:

HashSet : Hashtable qui implémente l'interface Set
TreeSet: arbre qui implémente l'interface SortedSet
ArrayList: tableau dynamique qui implémente l'interface List
LinkedList: liste doublement chaînée qui implémente l'interface List
HashMap: Hashtable qui implémente l'interface Map
TreeMap: arbre qui implémente l'interface SortedMap.

Parallèlement, le framework définit des interfaces pour faciliter le parcours des collections et leur tri:

Iterator : interface pour le parcours des collections,
ListIterator : interface pour le parcours des listes dans les deux sens et modifier les éléments lors de ce parcours
Comparable: interface pour définir un ordre de tri naturel pour un objet
Comparator: interface pour définir un ordre de tri quelconque

II Schéma du framework collections (2/2)



II Concepts de collections

Qu'est ce qui est stocké dans les collections ?

Une collection peut stocker des éléments de type quelconque, mais ces éléments sont *obligatoirement* *des objets.*

Par exemple, un vecteur dynamique peut stocker des chaînes (String), des objets de type Float, Integer, des objets de type Compte, de type Point…

Mais il est très délicat de manipuler de telles collections, vu qu'il faudra très souvent recourir au casting (opérateur de cast) et à instanceof (pour trouver la classe d'un objet).

Donc il est préférable d'essayer de stocker dans une collection des objets de même type pour faciliter leur manipulation.

Dans l'introduction d'un élément dans une collection , on ne réalise pas de recopie d'objet, on se contente en fait d'introduire la référence à l'objet.

Ainsi, il est possible d'introduire la référence null dans une collection

L'interface java.util.Collection

Cette interface définit des méthodes pour des objets qui gèrent des éléments d'une façon assez générale.

*/*ajoute l'élément fourni en paramètre à la collection. La valeur de retour indique si la collection a été mise à jour*/*

boolean add (Object)

*/*ajoute à la collection tous les éléments de la collection fournie en paramètre*/*

boolean addAll (Collection)

*/*supprime tous les éléments de la collection*/*

void clean ( )

*/*indique si la collection contient au moins un élément identique à celui fourni en paramètre (au sens de equals)*/*

boolean contains (Object)

*/*indique si tous les éléments de la collection fournie en paramètre sont*

boolean containsAll (Collection)

```
/*indique si la collection est vide*/
 boolean isEmpty ( )
/*renvoie un itérateur ie un objet qui permet de parcourir l'ensemble des
 éléments de la collection*/
Iterator iterator ( )
/*supprime l'élément fourni en paramètre de la collection. La valeur de
retour indique si la collection a été mise à jour*/
 boolean remove (Object)
/*supprime tous les éléments de la collection qui sont contenus dans la
 collection*/
 boolean removeAll (Collection)
/*renvoie le nombre d'éléments contenus dans la collection*/
 int size ( )
```

L'interface java.util.Iterator

Cette interface fournit des méthodes pour des objets capables de parcourir les données d'une collection:

```
 boolean hasNext ( ) // indique si il reste à parcourir dans la collection
Object next ( )       // renvoie le prochain élément dans la collection
 void remove ( )      // supprime le dernier élément parcouru.
```

Remarque: la méthode next ( ) lève une exception de type java.util.NoSuchElementException, si elle est appelée alors que la fin du parcours des éléments est atteinte. Pour éviter de lever cette exception, il suffit d'appeler la méthode hasNext ( ) avec le résultat de l'appel à la méthode next ( ).

```
/* construit un itérateur monodirectionnel sur la collection*/
     Iterator   iter = objetcollection.iterator ( ) ;
/*vérifie si l'itérateur est ou non en fin de collection*/
while (iter. hasNext ( ))
  {    /*renvoie l' Object désigné par l'itérateur */
       Object o = iter.next ( ) ;
       System.out.println (" objet"+  o);
  }
```

NB: la méthode remove ( ) supprime l'élément renvoyé par le dernier appel de next( ). Ainsi, il est impossible d'appeler remove( ) sans un appel correspondant de la méthode next( ). On emploie remove ( ) comme suit:

```
iter.next ( ) ;       // Si aucun appel à next( ) ne correspond à celui de remove( ) ,
iter.remove( ) ;  // une exception de type IllegalStateException est levée.
```

Cette interface définit un itérateur bidirectionnel en disposant de méthodes qui permettent de parcourir certaines collections (vecteurs dynamiques, listes chaînées) dans les deux sens et d'effectuer des mises à jour qui agissent par rapport à l'élément courant dans le parcours.

Cette interface dispose d'une méthode nommée **listIterator** qui fournit un objet qui est un itérateur bidirectionnel.

Il s'agit ici d'objet d'un type implémentant l'interface **ListIterator** (dérivée de Iterator)

Cet objet dispose donc des méthodes **next ( )**, **hasNext( )** et **remove ( )** mais aussi il dispose d'autres méthodes permettant d'exploiter le caractère bidirectionnel:

```
void add( Object)        // ajoute un élément dans la liste à la position courante
boolean hasPrevious( ) // indique si il reste au moins un élément à parcourir dans la
                        // liste dans son sens inverse
Object previous( )      // renvoie l'élément précédent dans la liste
void set (Object)       // remplace l'élément courant par celui en paramètre
```

On suppose qu'on dispose d'un objet l de type LinkedList

```
/* construit un itérateur bidirectionnel en fin de liste sur l'objet l*/
    ListIterator  iter = l.listIterator ( l.size()) ;
/*vérifie si l'itérateur est ou non en fin de collection dans le sens inverse*/
while (iter. hasPrevious ( ) )
  {   /*renvoie l' Object précédent désigné par l'itérateur */
      Object o = iter.previous ( ) ;
      System.out.println (" objet"+  o);
  }
```

NB: L'interface Iterator ne dispose pas de méthode d'ajout d'un élément à une position donnée.

L'interface java.util.List

Une liste est une collection ordonnée d'éléments qui autorise d'avoir des doublons.
Étant ordonné, un élément d'une liste peut être accédé à partir de son index.

L'interface List étend l'interface Collection.
L'interface est renforcée par des méthodes permettant d'ajouter ou de retirer des éléments se trouvant à une position donnée
Les collections qui implémentent cette interface autorisent les doublons dans les éléments de la liste. Elles autorisent aussi l'insertion d'éléments null.
L'interface List propose plusieurs méthodes pour un accès aux éléments à partir d'un index.
La gestion de cet index commence par zéro.

Pour les listes, une interface particulière est définie pour assurer le parcours de la liste dans les deux sens et effectuer des mises à jour: l'interface ListIterator.

```
ListIterator listIterator( )  // renvoie un objet pour parcourir la liste
Object set( int i, Object o)  //remplace l'élément  à l'indice i par l'objet o
void add( int i, Object o)    // ajoute à la liste l'objet o à la position i
Object get (int )              // renvoie l'objet à la position spécifiée
 int indexOf (Object o)       //renvoie la position du premier o trouvé et -1 si l'élément
                              // n'est pas dans la liste
List subList (int, int)       /*renvoie un extrait de la liste contenant les éléments
                               entre les deux index fournis (le premier inclus, le dernier
                               exclu); les éléments contenus dans la liste de retour sont
                              des références sur la liste originale; des mises à jour de
                              cette liste impactent la liste originale*/
 int lastIndexOf (Object)     //renvoie l'index du dernier élément fourni en paramètre
                               // ou -1 si l'élément n'est pas dans la liste
```

Le framework propose des classes qui implémentent l'interface List: LinkedList et ArrayList.

Les listes chaînées: la classe java.util.LinkedList

Cette classe hérite de AbstractSequentialList et implémente donc l'interface List.
Elle représente une liste doublement chaînée (introduction d'une référence à l' élément suivant et une référence à l'élément précèdent).
On peut réaliser une pile à partir d'une *LinkedList*.

Les constructeurs

```
LinkedList ( )                // pour créer une liste vide
LinkedList ( Collection c)    // créer une liste initialisée à partir des éléments
                              // de la collection c
```

```
import java.util .*; // package de base des collections
public class ExempleLinkedList01 {
  public static void main (String [ ] args) {
  LinkedList <String> list = new LinkedList <String> ( ) // cree une liste vide
  /*ajout d elements dans la liste*/
  list.add ("objet 1") ;
  list.add ("objet 2") ;
  list.add ("objet 3") ;
  /* iterateur bidirectionnel sur la liste*/
  ListIterator <String> iter = list.listIterator ( ) ;
  /*parcours des elements de la liste*/
  while (iter.hasNext ( ) )
    { String o = iter.next ( )  ;
      System.out .println ("element "+o) ;
    }
  }
}
```

Commentaires sur l'exemple de liste chaînée

Une liste chaînée gère une collection de façon ordonnée: l'ajout d'un élément peut se faire en fin de liste ou après n'importe quel élément. Dans ce cas l'ajout est lié à la position courante lors du parcours.

Les iterator peuvent être utilisés pour faire des mises à jour de la liste: une exception de type ConcurrentModificationException est levée si un iterator parcourt la liste alors q'un autre fait des mises à jour (ajout ou suppression d'un élément de la liste).

Pour gérer facilement cette situation, il faut mieux disposer d'un seul iterator s'il y a des mises à jour prévues dans la liste.

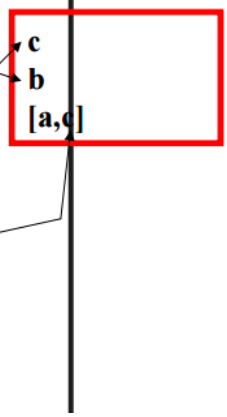
Voici quelques méthodes spécifiques de la classe LinkedList:

```
void addFirst( Object)      // ajoute l'élément en début de liste
void addLast (Object)       // ajoute l'élément en fin de liste
Object getFirst ( )         // renvoie le premier de la liste
Object getLast ( )          // renvoie le dernier de la liste
Object removeFirst ( )      // supprime et renvoie le premier élément
Object removeLast ( )       // supprime et renvoie le dernier élément
```

La méthode toString ( ) renvoie une chaîne contenant tous les éléments de la liste.

Il n'existe pas de méthode pour accéder directement à un élément de la liste.

```
import java.util .*; // package de base des collections
public class ExempleLinkedList02 {
  public static void main(String[] args) {
  LinkedList <String> list = new LinkedList <String> ( ); // cree une liste vide
  /*ajout d elements dans la liste*/
  list.add (" a") ;     list.add (" b") ;      list.add (" c") ;
  /* iterateur bidirectionnel sur la liste//on se place en fin de liste*/
  ListIterator <String> iter = list.listIterator (list.size( )) ;
  /*parcours des elements de la liste en sens inverse*/
  while (iter.hasPrevious ( ) )
    { String  s = iter.previous ( )  ; System.out .println ( s );
      if (s.equals ("b ")) ;
                          {   iter.remove ( );  break; }
    }
System.out .println( list. toString( )  ) ; // affiche les éléments de la liste
```



Les Tableaux redimensionnables: la classe java.util.ArrayList

Cette classe représente un tableau d'objets dont la taille est dynamique.
Elle hérite de la classe AbstractList donc elle implémente l'interface List.

Le fonctionnement de cette classe est identique à celle de la classe Vector.
La différence avec la classe Vector est que cette dernière est multi-thread (toutes les méthodes sont synchronisées).
Pour une utilisation dans un seul thread, la synchronisation des méthodes est inutile et coûteuse. Il est donc préférable d'utiliser un objet de la classe ArrayList.

Les constructeurs

```
ArrayList   ( )                  // vecteur dynamique vide
ArrayList (Collection c) // vecteur dynamique contenant tous les éléments de c
```

ArrayList: les méthodes

```
boolean add (Object )              // ajoute un element en fin de tableau
boolean addAll (Collection )       // ajoute tous les éléments de la collection en fin de tableau
boolean addAll (int, Collection )  // ajoute tous les éléments de la collection à partir de la
                                   // position indiquée
void clear ( )                     // supprime tous les éléments du tableau
void ensureCapacity (int)          //permet d'augmenter la capacité du tableau pour s'assurer
                                   // qu'il puisse contenir le nombre d'éléments passé en paramètre
Object get( int)                   // renvoie l'élément du tableau dont la position est précisée
int indexOf (Object  o)            // renvoie la position de la première occurrence de l'élément o
boolean isEmpty ( )                // indique si le tableau est vide
int lastIndexOf (Object o)         // renvoie la position de la dernière occurrence de l'élément o
Object remove (int)                // supprime dans le tableau l'élément dont la position est indiquée
void removeRange (int i ,int j)    //supprime tout élément  entre i (inclus) et j (exclu)
Object set (int, Object)           // remplace l'élément à la position indiquée par celui en paramètre
int size ( )                       // renvoie le nombre d'élément du tableau
```

ArrayList: quelques remarques

Chaque objet ArrayList gère une capacité qui est le nombre d'éléments qu'il est possible d'insérer avant d'agrandir le tableau. Cette capacité a une relation avec le nombre d'élément de la collection.
Lors de l'ajout, cette capacité et le nombre d'élément de la collection détermine si le tableau doit être agrandi.
L'agrandissement de cette capacité s'effectue avec la méthode ensureCapacity(int) .

Exemple

```
public class TestArrayList {
 public static void main(String[] args) {
  ArrayList c = new ArrayList();
 /*ajout de dix element de type Integer*/
 for (int i = 0;i < 10;i++)  c.add (new Integer(i)) ;
 /*affiche des element par accès direct*/
 for (int i = 0;i < c.size() ;i++)
  System.out .print( c. get(i) +" "   );   (1)
  System.out .println();
/* suppression d'element aux positions indiquées*/
c. remove(2) ;  c. remove(4) ;  c. remove(6) ;  // c.remove(8)
/*affiche de la liste */
System.out .print (c.toString()   );   (2)
/*ajout de dix element de type Integer*/
c.add (2,"100") ;c.add(2,"200") ;c.add(5,"300") ;
 /*réaffiche de la liste */
System.out .print(c.toString()   );   (3)
```

Résultats:
0 1 2 3 4 5 6 7 8 9 (1)
[0, 1, 3, 4, 6, 7, 9] (2)
[0, 1, 200, 100, 3, 300, 4, 6, 7, 9] (3)

Après suppression de 2, 5 et 8 c.size() = 7 donc si vous faîtes c.remove (8); vous obtenez Une exception ArrayIndexOutOfBounds - -Exception.

Exemple :

L'interface java.util.Map

Ce type de collection désignée sous le titre de table associative gère les informations sous forme de ***paires*** clé/valeur. Cette interface n'autorise pas de doublons.
L'intérêt des tables associatives est de pouvoir retrouver rapidement une ***clé*** donnée pour en obtenir l'information associée qui est ***sa valeur***.
Deux types d'organisations sont rencontrées avec les ensembles:

- table de hachage: classe **HashMap**, **Hashtable**.
- arbre binaire: classe **TreeMap**.

Seule la clé est utilisée pour ordonnancer les informations :

pour ***HashMap*** on se servira du code de hachage des objets formant les clés;
pour ***TreeMap***, on se servira de la relation d'ordre induite par **compareTo** ou par un comparateur fixé à la construction.

interface java.util.Map: les méthodes

```
void clear( )                          // supprime tous les éléments de la collection
boolean containsKey ( Object o)        // indique si la clé o est contenue dans la collection
boolean containsValue (Object o)       //indique si la valeur o est dans la collection
Set entrySet ( )          // renvoie un ensemble contenant toutes les valeurs de la collection
Object get (Object o) // renvoie la valeur associée à la clé o
boolean isEmpty( )                     // indique si la collection est vide
Set keySet( )                          // renvoie un ensemble contenant les clés de la collection
Object put ( Object c, Object v)       // insère la clé c et sa valeur v dans la collection
void putAll ( Map m)                   // insère toutes les clés/valeurs dans la collection
Collection values ( )     // renvoie tous les éléments de l'ensemble dans une collection
Object remove (Object o) // supprime l'élément de clé o
int size ( )                  // renvoie le nombre d'élément de la collection
```

java.util.HashMap

La classe HashMap n'est pas synchronisée; pour assurer la gestion des accès concurrents sur cet objet, il faut l'envelopper dans un objet Map en utilisant la méthode **synchronizedMap** de l'interface Collection.

```
    public static void main(String[] args) {
        Map<String, Fruit> lesFruits=new HashMap<String,Fruit>();
        lesFruits.put("o1", new Orange(60));
        lesFruits.put("p1", new Pomme(60));
        lesFruits.put("p2", new Pomme(70));

        lesFruits.get("p1").affiche();

        Set<String> keys=lesFruits.keySet();
        for(String cle:keys){
            lesFruits.get(cle).affiche();
        }

        Collection<Fruit> values=lesFruits.values();

        for(Fruit f:values){
            f.affiche();
        }
    }

}
```