Les collections en Java

Avant Java2 qui a introduit l'API **Collections**, seules quelques classes du package java.util permettaient de stocker et de gérer des éléments : Array, Vector, Stack, Hashtable, Properties et BitSet. L'interface Enumeration permet de parcourir le contenu de ces objets.

A partir de La version 2 de Java il y a eu l'élargissement de la bibliothèque de classes utilitaires (java.util). On y trouve désormais des classes permettant de manipuler les principales structures de données, c'est-à-dire les vecteurs dynamiques, les ensembles, les listes chaînées, les queues et les tables associatives.

C'est ainsi que les classes relatives aux vecteurs, aux listes, aux ensembles et aux queues implémentent une même interface (Collection) qu'elles complètent de fonctionnalités propres.

On ne verra pas toutes les classes Collection car elles sont nombreuses, mais nous verrons les principales d'entre elles.

Nous étudierons ensuite en détail chacune de ces structures, à savoir :

- les listes, implémentées par la classe LinkedList,
- les vecteurs dynamiques, implémentés par les classes ArrayList et Vector,
- les ensembles, implémentés par les classes HashSet et TreeSet,
- Nous terminerons enfin par les tables associatives qu'il est préférable d'étudier séparément des autres collections car elles sont de nature différente (notamment, elles n'implémentent plus l'interface Collection mais l'interface Map).

I. Les différentes Interfaces et implémentations

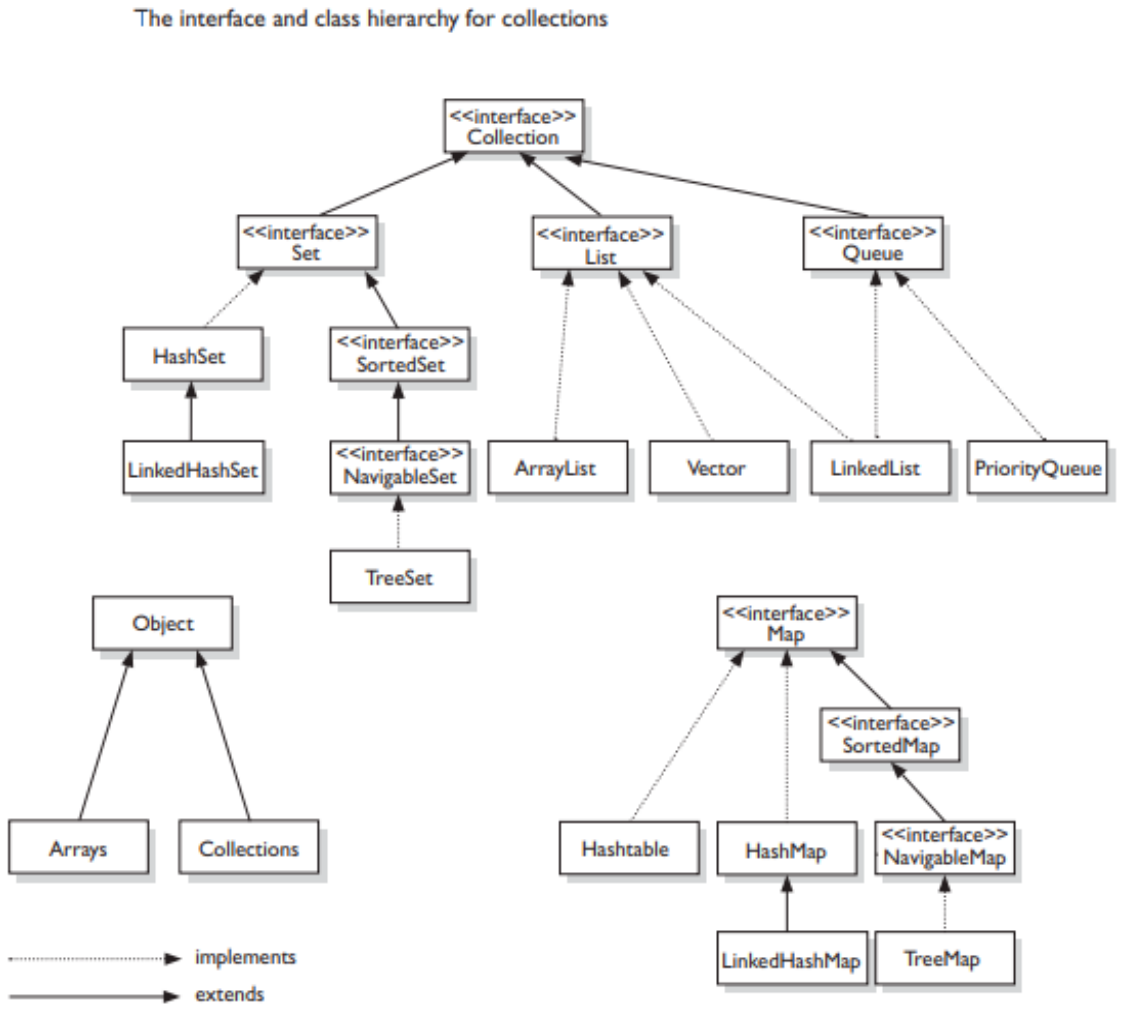
Il existe deux types de collections :

- Les collections **séquentielles** : classes relatives aux vecteurs, listes, ensembles et aux queues implémentent une même interface (Collection)
- Les collections **associatives** : Les dictionnaires, tables associatives (implémentent l'interface Map)

Avant d'étudier certaines classes, ci-dessous la hiérarchie d'interfaces composant ce qu'on appelle les collections. Ces interfaces encapsulent la majeure partie des méthodes utilisables.

Il faut aussi ajouter que toutes ces interfaces utilisent des génériques et chaque interface a ses avantages et inconvénients. Donc selon nos besoins il faut choisir la meilleure collection.

Hiérarchie d'interfaces :



II. L'interface Collection

L'interface Collection, ajoutée à Java 1.2, définit des méthodes pour des objets qui gèrent des éléments d'une façon assez générale. Elle est la super interface de plusieurs interfaces de la bibliothèque de Java. Ci-dessous ses différentes méthodes les plus utilisées :

Méthode	Rôle
boolean **add**(E e)	Ajouter un élément à la collection
boolean **addAll** (Collection c)	Ajoute tous les éléments de c à la collection en cours.
void **clear**()	Supprime tous les éléments de la collection
boolean **contains**(Object o)	Retourner un booléen qui précise si l'élément est présent dans la collection
boolean **containsAll** (Collection c)	Retourne un booléen qui précise si la collection c est incluse dans la collection en cours.

boolean **isEmpty**()	Retourne un booléen qui précise si la collection est vide
Iterator<E> **iterator**()	Retourne un Iterator qui permet le parcours des éléments de la collection
boolean **remove**(Object o)	Supprime un élément de la collection s'il est présent (optionnelle)
boolean **removeAll** (Collection c)	supprime de la collection tous les éléments présents dans c
boolean **retainAll** (Collection c)	ne conserve dans la collection que les éléments présents dans c
int **size**()	Retourne le nombre d'éléments contenus dans la collection
Object[] **toArray**()	Retourne un tableau contenant tous les éléments de la collection

2.1 Les itérateurs

Les itérateurs sont des objets qui permettent de "parcourir" un par un les différents éléments d'une collection.

Il existe deux sortes d'itérateurs :

- monodirectionnels : le parcours de la collection se fait d'un début vers une fin ; on ne passe qu'une seule fois sur chacun des éléments ;
- bidirectionnels : le parcours peut se faire dans les deux sens ; on peut avancer et reculer à sa guise dans la collection.

2.2 Les itérateurs monodirectionnels : l'interface Iterator

Chaque classe collection dispose d'une méthode nommée iterator fournissant un itérateur monodirectionnel, c'est-à-dire un objet d'une classe implémentant l'interface Iterator<E> (Iterator avant le JDK 5.0). Associé à une collection donnée.

À un instant donné, un itérateur indique une position courante désignant soit un élément donné de la collection, soit la fin de la collection. Comme on peut s'y attendre, le premier appel de la méthode iterator sur une collection donnée fournit comme position courante, le début de la collection.

Ci-dessous les différentes méthodes de l'interface Iterator :

Méthode	**Description**
boolean **hasNext**()	Test si l'itérateur est ou non en fin de collection

E **next**()	Avance l'itérateur d'une position et retourne l'élément qui se trouve à cette position.
void **remove**()	Supprime l'élément courant

2.3 Les itérateurs bidirectionnels : l'interface ListIterator

Certaines collections (listes chaînées, vecteurs dynamiques) peuvent, par nature, être parcourues dans les deux sens. Elles disposent d'une méthode nommée **listIterator** qui fournit un itérateur bidirectionnel. Il s'agit, cette fois, d'objet d'un type implémentant l'interface **ListIterator<E>** (**dérivée de Iterator<E>).** Il dispose bien sûr des méthodes next, hasNext et remove héritées de Iterator.

Mais il dispose aussi d'autres méthodes permettant d'exploiter son caractère bidirectionnel, à savoir : les méthodes **previous** et **hasPrevious**, complémentaires de **next** et **hasNext**, mais aussi, des méthodes d'addition d'un élément à la position courante (add) ou de modification de l'élément courant (set).

Ci-dessous les différentes méthodes de l'interface ListIterator :

Méthode	**Description**
void **add**(E e)	Ajoute un élément dans la collection
boolean **hasPrevious**()	Test si l'élément précédent existe
boolean **hasNext**()	Test si l'élément suivant existe
E **previous**()	Retourne l'élément précédent
E **next**()	Retourne l'élément suivant
void **set**(E e)	Remplace l'élément courant par celui fourni en paramètre
void **remove**()	Supprime l'élément courant

2.4 Exemple d'application

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;
class TestIterator{
   public static void main(String args[]){
     // Déclaration d'une collection générique avec
     //les éléments de type String
      Collection<String>MaCollection ;
      // Création d'un ArrayList pour Les String
       MaCollection  =new ArrayList<String>();
    //Ajout des élements dans cette collection
```

```
        MaCollection.add("elt1");
        MaCollection.add("elt2");
        MaCollection.add("elt3");
//Mettre le contenu de la collection dans it pour le
//parcourir avec l'énumérateur
Iterator<String> it=MaCollection.iterator();
        String elt;
        while(it.hasNext())
         {elt=it.next();
         System.out.println(elt);
         if(elt.equals("elt2"))
            it.remove();
         }
        System.out.println(MaCollection);
    }
}
```

Le résultat de l'exécution est :

```
elt1
elt2
elt3
[elt1, elt3]
```

III. L'interface List

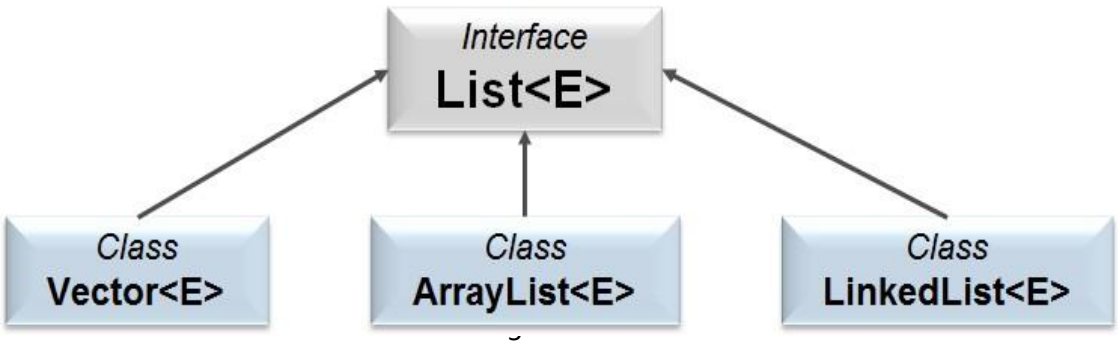
3.1 Définition

L'interface List représente une collection ordonnée de données qui peuvent être des doublons (données identiques) et elle peut accepter null comme valeur de ses éléments.

Elle peut apparaître comme un tableau qui n'a pas la contrainte de taille pour l'ajout des éléments.

En plus des actions globales à l'interface Collection, l'interface List permet également :

- d'accéder à une donnée à partir de sa position (index) ;
- de récupérer la position d'un objet dans la liste.



3.2 Les méthodes usuelles de l'interface List

Les méthodes usuelles sont les suivantes :

Méthode	Description
void **add**(E e)	Ajoute un élément à la fin de la liste
void **add** (int index, E elem)	ajoute elem à la position index
boolean **addAll** (int index, Collection c)	ajoute tous les éléments de c à partir de la position index
E **get**(int i)	Retourne l'élément de à la position i
int **indexOf** (Object elem)	fournit le rang du premier élément valant elem
int **lastIndexOf** (Object elem)	fournit le rang du dernier élément valant elem
ListIterator **listIterator** ()	fournit un itérateur bidirectionnel
ListIterator **listIterator** (int index)	crée un itérateur, initialisé à index
int **size**()	Retourne la taille de la liste
void **clear**()	Supprime tous les éléments de tableau
E **set** (int index, E elem)	remplace l'élément de rang index par elem
void **add**(int i, E e)	Insert l'élément e à la position i
E **remove** (int index)	supprime l'objet de rang index et en fournit la valeur
Object[] **toArray**()	Renvoie un tableau contenant tous les éléments de la liste
List **subList** (int debut, int fin)	fournit une vue d'une sous-liste de la liste d'origine, constituée des éléments de rang début à fin

3.3 La classe ArrayList

- **Définition et avantages**

La classe Java **ArrayList** est une classe qui implémente l'interface List. Elle utilise un tableau dynamique pour stocker les éléments.
Elle offre des fonctionnalités d'accès rapide comparables à celles d'un tableau d'objets. Sa mise en œuvre est prévue pour permettre des **accès** efficaces à un élément de rang donné (get) et des modifications efficaces aussi (set).
Cette classe offre plus de souplesse que les tableaux d'objets dans la mesure où sa taille (son nombre d'éléments) peut varier au fil de l'exécution.

Inconvénients

Pour que l'accès direct à un élément de rang donné soit possible, il est nécessaire que les références des objets soient contiguës en mémoire (à la manière de ceux d'un tableau).

Aussi cette classe souffrira d'une lacune inhérente à sa nature : l'insertion ou la suppression d'un objet à une position donnée ne pourra plus se faire aussi **rapidement** que dans le cas d'une liste chaînée puisque toutes les cases suivantes du tableau devront être décalées. Donc la manipulation est lente car de nombreux déplacements doivent avoir lieu si un élément est supprimé de la liste.

Les méthodes les plus utilisées de la classe ArrayList

Méthode	Description
ArrayList ()	Il est utilisé pour construire une liste de tableau vide.
ArrayList(Collection<? extends E> c)	Construire une liste de tableaux qui est initialisée avec les éléments de la collection c.
ArrayList(int capacity)	créer une liste de tableaux ayant la capacité initiale spécifiée.
void **add**(int index, E element)	Il est utilisé pour insérer l'élément spécifié à la position spécifiée dans une liste.
boolean **add**(E e)	Il est utilisé pour ajouter l'élément spécifié à la fin d'une liste.
boolean **addAll** (Collection<? extends E> c)	Il est utilisé pour ajouter tous les éléments de la collection spécifiée à la fin de cette liste, dans l'ordre dans lequel ils sont renvoyés par l'itérateur de la collection spécifiée.
boolean **addAll**(int index, Collection<? extends E> c)	Il est utilisé pour ajouter tous les éléments de la collection spécifiée, en commençant à la position spécifiée de la liste.
void **clear**()	Il est utilisé pour supprimer tous les éléments de cette liste.
E **get**(int index)	Il est utilisé pour extraire l'élément de la position particulière de la liste.

boolean **isEmpty**()	Il renvoie true si la liste est vide, sinon false.
int **lastIndexOf**(Object o)	Il est utilisé pour renvoyer l'index dans cette liste de la dernière occurrence de l'élément spécifié, ou -1 si la liste ne contient pas cet élément.
Object[] **toArray**()	Il est utilisé pour renvoyer un tableau contenant tous les éléments de cette liste dans le bon ordre.
boolean **contains**(Object o)	Il renvoie true si la liste contient l'élément spécifié
int **indexOf**(Object o)	Il est utilisé pour renvoyer l'index dans cette liste de la première occurrence de l'élément spécifié, ou -1 si la liste ne contient pas cet élément.
E **remove**(int index)	Il est utilisé pour supprimer l'élément présent à la position spécifiée dans la liste.
boolean **remove**(Object o)	Il est utilisé pour supprimer la première occurrence de l'élément spécifié.
boolean **removeAll**(Collection<?> c)	Il est utilisé pour supprimer tous les éléments de la liste.
protected void **removeRange** (int fromIndex, int toIndex)	Il est utilisé pour supprimer tous les éléments situés dans la plage donnée.
void **retainAll**(Collection<?> c)	Il est utilisé pour conserver tous les éléments de la liste présents dans la collection spécifiée.
E **set**(int index, E element)	Il est utilisé pour remplacer l'élément spécifié dans la liste, présent à la position spécifiée.

void **sort**(Comparator<? super E> c)	Il est utilisé pour trier les éléments de la liste sur la base du comparateur spécifié.
List<E> **subList**(int fromIndex, int toIndex)	Il est utilisé pour récupérer tous les éléments compris dans la plage donnée.
int **size**()	Il est utilisé pour renvoyer le nombre d'éléments présents dans la liste.

- **Exemple d'application**

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.ListIterator;
 class Main {
   public static void main(String[] args) {
     //Déclaration et creation d'un Arraylist composé de String
     List<String> list = new ArrayList<String>();
     // Ajout des élements dans la collection list
     list.add("a");
     list.add("b");
     list.add("c");
     list.add("d");
     list.add("e");
     list.add("f");

     //On met la liste dans le désordre
     Collections.shuffle(list);
     System.out.println(list);

     //On la remet dans l'ordre
     Collections.sort(list);
     System.out.println(list);

     Collections.rotate(list, -1);
     System.out.println(list);

     //On récupère une sous-liste
     List<String> sub = list.subList(2, 5);
     System.out.println(sub);
     Collections.reverse(sub);
     System.out.println(sub);
```

```
    //On récupère un ListIterator
    ListIterator<String> it = list.listIterator();
    while(it.hasNext()){
      String str = it.next();
      if(str.equals("d"))
        it.set("z");
    }
    while(it.hasPrevious())
      System.out.print(it.previous());
    System.out.println();
    }
}
```

Exécution

```
[a, c, e, d, b, f]
[a, b, c, d, e, f]
[b, c, d, e, f, a]
[d, e, f]
[f, e, d]
azefcb
```

3.4 La classe LinkedList

Définition et avantages

La classe **LinkedList** est une classe qui implémente l'interface List. Elle permet de manipuler des listes dites "doublement chaînées". À chaque élément de la collection, on associe deux informations qui sont les références à l'élément précédent et au suivant.

Une telle collection peut ainsi être parcourue à l'aide d'un itérateur bidirectionnel de type **ListIterator**.

Le grand avantage d'une telle structure est de permettre des ajouts ou des suppressions à une position donnée (ceci grâce à un simple jeu de modification de références) donc **aucun décalage n'est nécessaire**.

La classe Java LinkedList peut être utilisée en tant que liste, pile ou file d'attente.

Inconvénients

En revanche, l'accès à un élément en fonction de sa valeur ou de sa position dans la liste sera peu efficace puisqu'il nécessitera obligatoirement de parcourir une partie de la liste. Donc **mauvaise performance** pour lecture et écriture

Les méthodes les plus utilisées de la classe LinkedList

Méthode	Description
LinkedList ()	Construire une liste vide
LinkedList(Collection<? extends E> c	Construire une liste contenant les éléments de la collection spécifiée. Dans l'ordre, ils sont renvoyés par l'itérateur de la collection.
void addFirst(E e)	Il est utilisé pour insérer l'élément donné au début d'une liste.
void **addLast**(E e)	Il est utilisé pour ajouter l'élément donné à la fin d'une liste.
Iterator<E> **descendingIterator**()	Il est utilisé pour renvoyer un itérateur sur les éléments dans un ordre séquentiel inverse.
E **element**()	Il est utilisé pour récupérer le premier élément d'une liste.
E **getFirst**()	Il est utilisé pour renvoyer le premier élément d'une liste
E **getLast**()	Il est utilisé pour renvoyer le dernier élément d'une liste.
boolean **offer**(E e)	Il ajoute l'élément spécifié en tant que dernier élément d'une liste.
boolean **offerFirst**(E e)	Il insère l'élément spécifié au début d'une liste.
boolean **offerLast**(E e)	Il insère l'élément spécifié à la fin d'une liste.
E **peek**()	Il récupère le premier élément d'une liste
E **peekFirst**()	Il récupère le premier élément d'une liste ou renvoie null si une liste est vide.

E **peekLast**()	Il récupère le dernier élément d'une liste ou renvoie null si une liste est vide.
E **poll**()	Il récupère et supprime le premier élément d'une liste.
E **pollFirst**()	Il récupère et supprime le premier élément d'une liste ou renvoie la valeur null si une liste est vide.
E **pollLast**()	Il récupère et supprime le dernier élément d'une liste ou renvoie la valeur null si une liste est vide.
E **pop**()	Il récupère et supprime un élément de la pile représentée par une liste.
void **push**(E e)	Il pousse un élément sur la pile représentée par une liste.
E **remove**()	Il est utilisé pour récupérer et supprimer le premier élément d'une liste.
E **removeFirst**()	Il supprime et retourne le premier élément d'une liste.
boolean **removeFirstOccurrence**(Object o)	Il est utilisé pour supprimer la première occurrence de l'élément spécifié dans une liste (lorsque vous parcourez la liste de haut en bas).
E **removeLast**()	Il supprime et retourne le dernier élément d'une liste.
boolean **removeLastOccurrence**(Object o)	Il supprime la dernière occurrence de l'élément spécifié dans une liste (lorsque vous parcourez la liste de haut en bas).

- **Exemple d'application**

```
import java.util.* ;
 class Liste1
{ public static void main (String args[])
{ LinkedList<String> l=                 new LinkedList<String>() ;
// LinkedList l= new LinkedList() ; <--       avant JDK 5.0
System.out.print ("Liste au début : ") ;
affiche (l) ;
l.add ("a") ; l.add ("b") ; // ajouts en fin de liste
System.out.print ("Liste en B : ") ;
affiche (l) ;
ListIterator<String> it=l.listIterator() ;
// LinkedList l= new LinkedList() ; <--       avant JDK 5.0
it.next() ; // on se place sur le premier élément cad en a et il va insérer
juste après le a
it.add ("c") ; it.add ("b") ; // et on ajoute deux éléments
System.out.print ("Liste en C : ") ;
affiche (l) ;
it=l.listIterator() ;
it.next() ; // on progresse d'un élément
it.add ("b") ; it.add ("d") ;               // et on ajoute deux éléments
System.out.print ("Liste en D : ") ;
affiche (l) ;
it=l.listIterator (l.size()) ;              // on se place en fin de liste
while (it.hasPrevious())                    // on recherche le dernier b
{ String ch=                                it.previous() ;
//  String ch= (String) it.previous() ; <--      avant JDK 5.0
if (ch.equals ("b"))
{ it.remove() ; // et on le supprime
  break ;}}
System.out.print ("Liste en E : ") ;
affiche (l) ;
it=l.listIterator() ;
it.next() ;
it.next() ;                                 // on se place sur le deuxième élément
it.set ("x") ;                              // qu'on remplace par "x"
System.out.print ("Liste en F : ") ;
affiche (l) ;}
public static void affiche (LinkedList<String> l)
// public static void affiche (LinkedList l) <-- avant JDK 5.0
{ ListIterator<String> iter=l.listIterator ();
// ListIterator iter l.listIterator (); <--avant JDK 5.0
```

```
while (iter.hasNext())
   System.out.print (iter.next() + " ") ;
System.out.println () ;
}}
```

Exécution:

```
Liste au début :
Liste en B : a b
Liste en C : a c b b
Liste en D : a b d c b b
Liste en E : a b d c b
Liste en F : a x d c b
```

IV. L'interface Set (Ensemble)

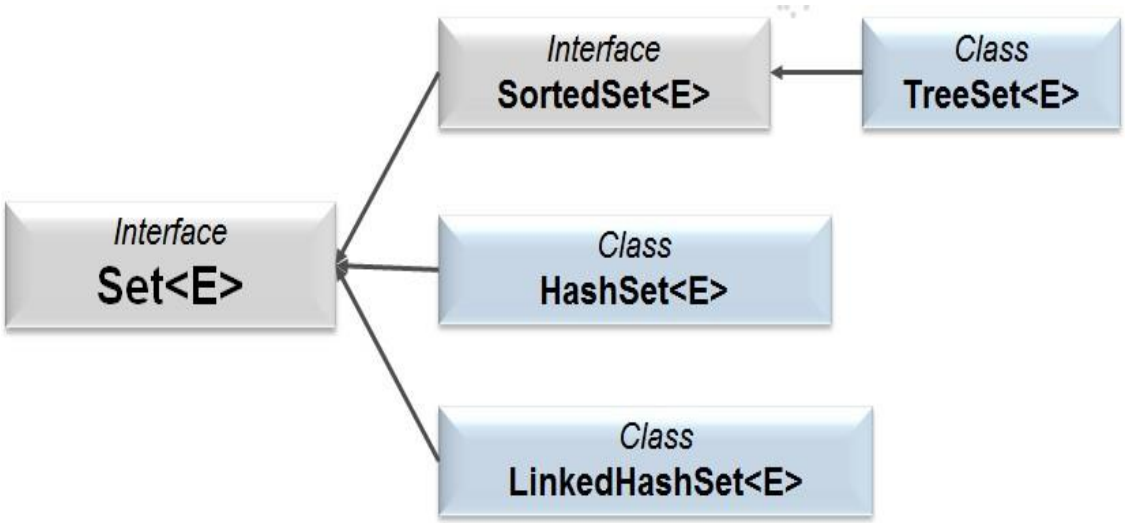
4.1 Définition

Un Set est une collection non ordonnée d'éléments, aucun élément ne pouvant apparaître plusieurs fois dans un même ensemble. Trois classes implémentent la notion d'ensemble : **HashSet , TreeSet et**

LinkedHashSet.

Pour les classes HashSet et TreeSet le test d'appartenance d'un élément dans chacune de ces collections aura besoin de certaines techniques :

- **HashSet** qui recourt à une technique dite de **hachage**
- **TreeSet** qui utilise un **arbre binaire** pour ordonner complètement les éléments.



4.2 Méthodes de l'interface Set

L'interface Set contient uniquement des méthodes héritées de Collection et ajoute la restriction selon laquelle les éléments en double sont interdits.

4.3 Les ensembles HashSet

Définition

La classe Java **HashSet** implémente l'interface Set. Elle est utilisée pour créer une collection qui utilise une table de hachage pour le stockage. La classe HashSet contient uniquement des éléments **uniques** et autorise la valeur **null**. Mais il faut noter qu'elle ne conserve pas l'ordre d'insertion puisque les éléments sont insérés sur la base de leur hashcode cad avec une table de hachage.

Fonctionnement de hachage

Ces tables de hachages sont un procédé qui permet de stocker des données : Première spécificité, les données sont gérées sous la forme clé-valeur. Deuxième spécificité, au lieu d'ajouter les données bout à bout (comme dans un tableau ou dans une List), les données sont éparpillées le plus uniformément possible.

Les données sont rangées en fonction de leur code de hachage (clé), retourné par la méthode hashcode() de chaque objet.

Donc suite à l'appel de la méthode **add** il y aura un appel implicite de la méthode **hashCode**() et selon la clé retournée par cette méthode il y aura l'ajout par ordre croissant de la clé.

Méthodes les plus utilisées de la classe HashSet

Certaines méthodes qui existent déjà dans la classe Object doivent être redéfinies :

- la méthode **equals** : c'est elle qui sert à définir **l'appartenance** d'un élément à l'ensemble puisque la méthode contains() qui existe déjà dans l'interface Collection se base sur la méthode equals.
- la méthode **hashCode** pour **ordonnancer** les éléments d'un ensemble, et on parlera de "table de hachage".

Les principales méthodes de cette classe sont les suivantes :

Méthode	Description
HashSet()	Il est utilisé pour construire un HashSet par défaut
HashSet(int capacity)	Il est utilisé pour initialiser la capacité du hachage défini avec la capacité de valeur entière donnée. La capacité augmente automatiquement à mesure que des éléments sont ajoutés au HashSet.
HashSet(Collection<? extends E> c)	Il est utilisé pour initialiser le hash set en utilisant les éléments de la collection c.
int **hashCode**()	Elle doit fournir le code de hachage correspondant à la valeur de l'objet.

- **Exemple d'application**

```
import java.util.* ;
class Point
{ private int x, y ;
   Point (int x, int y)
{ this.x=x ; this.y=y ; }
public int hashCode ()
{ return x+y ; }
public boolean equals (Object pp)
{ Point p=(Point) pp ;
return ((this.x==p.x) & (this.y==p.y)) ;}
public void affiche ()
{ System.out.print ("[" + x + " " + y + "] ") ;}}
 class EnsPoint
{ public static void main (String args[])
{ Point p1=new Point (1, 3), p2=new Point (0, 2) ;
Point p3=new Point (4, 5), p4=new Point (1, 7) ;
Point p[]={p1, p2, p1, p3, p4, p3} ;
HashSet<Point> ens=new HashSet<Point> () ;
for (Point px : p)
{ System.out.print ("le point ") ;
px.affiche() ;
boolean ajoute=ens.add (px) ;
if (ajoute) System.out.println ("a été ajouté ") ;
else System.out.println ("est déjà présent") ;
System.out.print ("ensemble= " ) ;
affiche(ens) ;}}

public static void affiche (HashSet<Point> ens)
{ Iterator<Point> iter=ens.iterator() ;
while (iter.hasNext())
{ Point p= iter.next() ;// Point p= (Point)iter.next() ; →avant JDK 5.0
  p.affiche() ;}
System.out.println () ;}}
```

L'éxécution est :

```
le point [1 3] a été ajouté
ensemble= [1 3]
le point [0 2] a été ajouté
ensemble= [0 2] [1 3]
le point [1 3] est déjà présent
ensemble= [0 2] [1 3]
```

```
le point [4 5] a été ajouté
ensemble= [0 2] [1 3] [4 5]
le point [1 7] a été ajouté
ensemble= [0 2] [1 3] [1 7] [4 5]
le point [4 5] est déjà présent
ensemble= [0 2] [1 3] [1 7] [4 5]
```

4.4 TreeSet

Définition et avantages

La classe **TreeSet** implémente l'interface **SortedSet** et cette dernière hérite de l'interface **Set**. Cette classe propose une autre organisation utilisant un "arbre binaire", lequel permet d'ordonner totalement les éléments.

On y utilise, la relation d'ordre usuelle induite par la méthode

compareTo des objets (de l'interface Comparable qu'on étudiera par la suite) ou par un comparateur (qu'on peut fournir à la construction de l'ensemble).

Dans ces conditions, la recherche dans cet arbre d'un élément de valeur donnée est généralement **moins rapide** que dans **une table de hachage mais plus rapide qu'une recherche séquentielle**. Par ailleurs, l'utilisation d'un arbre binaire permet de disposer en permanence d'un ensemble totalement ordonné (trié). On notera d'ailleurs que la classe TreeSet dispose de deux méthodes spécifiques first et last fournissant respectivement le premier et le dernier élément de l'ensemble.

Méthodes les plus utilisées de la classe TreeSet

Méthode	Description
TreeSet ()	Il est utilisé pour construire un ensemble d'arbres vide qui sera trié par ordre croissant en fonction de l'ordre naturel de l'ensemble.
TreeSet(Collection<? extends E> c)	Il est utilisé pour créer un nouvel ensemble d'arborescence contenant les éléments de la collection c.
E ceiling(E e)	Il renvoie l'élément égal ou le plus grand et le plus proche de l'élément spécifié dans l'ensemble, ou nul.

Comparator <?s u p e r E > **comparator**()	Il retourne le comparateur qui a classé les éléments dans l'ordre.
Iterator **descendingIterator**()	Itère les éléments dans l'ordre décroissant.
NavigableSet **descendingSet**()	Il retourne les éléments dans l'ordre inverse.
E **floor**(E e)	Elle renvoie l'élément égal ou le plus Petit et le plus proche de l'élément spécifié dans l'ensemble ou la valeur null, en l'absence d'un tel élément.
SortedSet **headSet**(E toElement)	Il renvoie le groupe d'éléments inférieur à l'élément spécifié.
E **pollFirst**()	Il est utilisé pour récupérer et supprimer le (premier) élément le plus bas.
E **pollLast**()	Il est utilisé pour récupérer et supprimer le plus haut (dernier) élément.
SortedSet **s u b S e t** (E fromElement, E toElement))	Il retourne un ensemble d'éléments compris entre la plage donnée, qui inclut fromElement et exclut toElement.
SortedSet **tailSet**(E fromElement)	Il renvoie un ensemble d'éléments supérieur ou égal à l'élément spécifié.
E **first**()	Il retourne le premier élément (le plus bas) actuellement dans cet ensemble trié.
E **last**()	Il retourne le dernier élément (le plus élevé) présent dans cet ensemble trié.

Les interfaces Comparable et Comparator

L'interface **Comparable** est utilisée pour ordonner les objets de la classe définie par l'utilisateur. Cette interface se trouve dans le package **java.lang** et ne contient qu'une méthode appelée **compareTo (E e).**

Certaines classes comme **String, File ou les classes enveloppes** (Integer, Float...) implémentent l'interface Comparable et disposent donc d'une méthode **compareTo** déjà implémentée
Sinon si les éléments d'une classes sont les objets d'une classe E qu'on va définir il faut lui faire implémenter l'interface Comparable et implémenter la méthode **compareTo.**

public int compareTo (E e) :

Elle est utilisée pour comparer l'objet actuel à l'objet spécifié. Elle retourne :

- entier positif, si l'objet actuel est supérieur à l'objet spécifié.
- entier négatif, si l'objet actuel est inférieur à l'objet spécifié.
- zéro, si l'objet actuel est égal à l'objet spécifié

Exemple d'application

```
import java.util.Iterator;
import java.util.TreeSet;

class Point implements Comparable
{ private int x, y ;
Point (int x, int y)
{ this.x=x ;
this.y=y ; }
@Override
public int compareTo (Object pp)
{ Point p=(Point) pp ;// egalite si coordonnees egales
if (this.x < p.x) return -1 ;
else if (this.x > p.x) return 1 ;
else if (this.y < p.y) return -1 ;
else if (this.y > p.y) return 1 ;
else return 0 ;
}
public void affiche ()
{ System.out.print ("[" + x + " " + y + "] ") ; }
}
class EnsPoint
{ public static void main (String args[])
{ Point p1=new Point (1, 3), p2=new Point (2, 2) ;
```

```
Point p3=new Point (4, 5), p4=new Point (1, 8) ;
Point p[]={p1, p2, p1, p3, p4, p3} ;
TreeSet<Point> ens=new TreeSet<Point> () ;
for (Point px : p)
{ System.out.print ("le point ") ;
px.affiche() ;
boolean ajoute=ens.add (px) ;
if (ajoute) System.out.println ("a été ajouté ") ;
else System.out.println ("est déjà présent") ;
System.out.print ("ensemble= " ) ;
affiche(ens) ;
}}
public static void affiche (TreeSet<Point> ens)
{ Iterator<Point> iter=ens.iterator() ;
while (iter.hasNext())
{ Point p= iter.next() ;
  p.affiche() ;}
System.out.println () ;
}}
```

L'exécution est:

```
le point [1 3] a été ajouté
ensemble= [1 3]
le point [2 2] a été ajouté
ensemble= [1 3] [2 2]
le point [1 3] est déjà présent
ensemble= [1 3] [2 2]
le point [4 5] a été ajouté
ensemble= [1 3] [2 2] [4 5]
le point [1 8] a été ajouté
ensemble= [1 3] [1 8] [2 2] [4 5]
le point [4 5] est déjà présent
ensemble= [1 3] [1 8] [2 2] [4 5]
```

L'interface Comparator

L'interface **Comparator** est utilisée pour trier les Collections. Les classes

implémentant l'interface **Comparator<E>** doivent implémenter la méthode :

int **compare**(E e1, E e2);

Exemple :

```
import java.util.Comparator;
class MyComparator implements Comparator<String>
{public int compare(String s1, String s2)
```

```
{   if(s1.length()==s2.length()) return 0;
   else if(s1.length()>s2.length())return 1;
   else return -1;
  }}
```

V. L'interface Map

5.1 Définition

Linterface Map ou table associative permet de conserver une information associant deux parties nommées clé et valeur. Une Map ne peut pas contenir de clés en double et chaque clé peut correspondre à au plus une valeur.

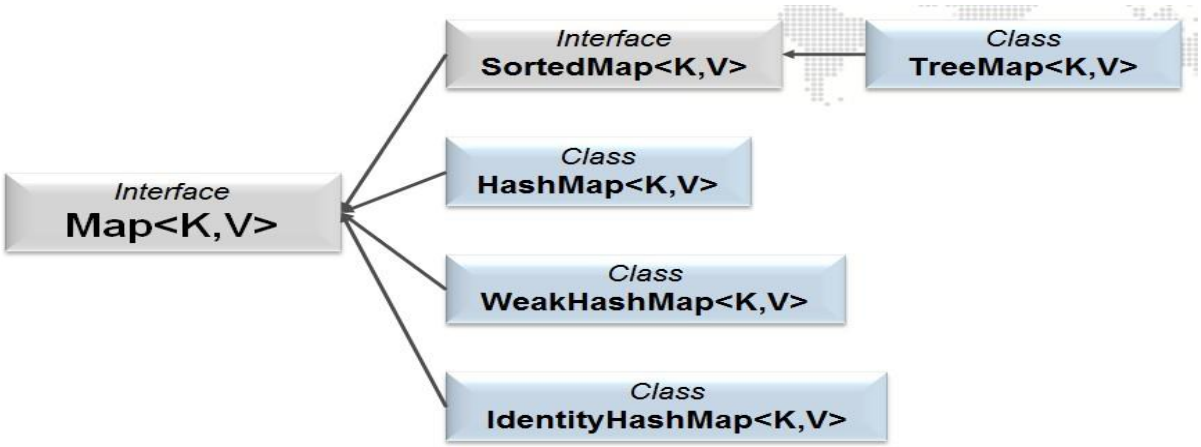
Elle est principalement destinée à retrouver la valeur associée à une clé donnée. Les exemples les plus caractéristiques de telles tables sont :

- ✓ Le dictionnaire : à un mot (clé), on associe une valeur qui est sa définition,
- ✓ L'annuaire usuel : à un nom (clé), on associe une valeur comportant le numéro de téléphone et, éventuellement, une adresse,
- ✓ L'annuaire inversé : à un numéro de téléphone (qui devient la clé), on associe une valeur comportant le nom et, éventuellement, une adresse.

On va donc tout naturellement retrouver les deux types d'organisation rencontrés pour les ensembles :

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

Dans les deux cas, seule la clé sera utilisée pour ordonnancer les informations.



5.2 La classe HashMap

Définition

La classe HashMap implémente l'interface **Map**. Elle utilise une table de hachage pour implémenter l'interface Map. Cela permet au temps d'exécution des opérations de base, telles que
get () et put (), de rester constant même pour les grands ensembles.

Une collection de type HashMap représente un tableau associatif : un ensemble de couples clé/valeur. La clé permet de retrouver rapidement la valeur qui lui a été associée. Les clés sont des objets uniques pouvant être null. Les valeurs sont des objets pouvant être redondants et null.

Dans une HashMap, il n'y a pas la notion d'ordre. On accède à ses éléments à l'aide de leur clé et non pas à l'aide de leur indice.

Méthodes usuelles de la classe HashMap :

Méthode	Description
HashMap ()	Ce constructeur construit un HashMap par défaut.
HashMap(Map m)	Ce constructeur initialise le HashMap en utilisant les éléments de l'objet Map m
HashMap(int capacity)	Ce constructeur initialise la capacité de la HashMap sur la valeur entière donnée, capacité.
void **clear**()	Supprime tous les mappages de cette carte.
Object **clone**()	Retourne une copie de cette instance HashMap.
boolean **containsKey**(Object key	Renvoie true si cette carte contient un mappage pour la clé spécifiée.
boolean **containsValue**(Object val)	Renvoie true si cette carte mappe une ou plusieurs clés sur la valeur spécifiée.
Set **entrySet**()	Renvoie une vue de collection des mappages contenus dans cette carte.
Object **get**(Object key)	Renvoie la valeur à laquelle la clé spécifiée est mappée dans cette HashMap, ou null si la Map ne

	contient aucun mappage pour cette clé.
Collection **values**()	Renvoie une vue de collection des valeurs contenues dans cette carte.
boolean **isEmpty**()	Renvoie true si cette Map ne contient aucune correspondance clé-valeur.
Set **keySet**()	Renvoie une vue d'ensemble des clés contenues dans cette Map.
Object **put** (Object key, Object value)	Associe la valeur spécifiée à la clé spécifiée dans cette Map.
putAll(Map m)	Copie tous les mappages de m vers cette Map. Ces mappages remplaceront tous les mappages que cette Map avait pour toutes les clés actuellement dans la carte spécifiée.
Object **remove**(Object key)	Supprime le mappage de cette clé de cette Map.
int **size**()	Renvoie le nombre de mappages clé-valeur dans cette carte.

- **Exemple d'application**

```
import java.util.* ;
class Map1
{ public static void main (String args[])
{ HashMap <String, String> m=  new HashMap <String, String>
() ;
m.put ("c", "10") ; m.put ("f", "20") ; m.put ("k", "30") ;
m.put ("x", "40") ; m.put ("p", "50") ; m.put ("g", "60") ;
System.out.println ("map initial :  " + m) ;
// retrouver la valeur associée à la clé "f"
String ch = m.get("f") ;
System.out.println ("valeur associée a f :  " + ch) ;

// ensemble des valeurs (attention, ici Collection, pas Set)
Collection<String> valeurs=  m.values () ;
System.out.println ("liste des valeurs initiales :  " + valeurs) ;
```

```
valeurs.remove ("30") ; // on supprime la valeur "30" par la vue associée
System.out.println ("liste des valeurs après sup :  " + valeurs) ;
// ensemble des clés
Set<String> cles=  m.keySet () ;
System.out.println ("liste des clés initiales :" + cles) ;
cles.remove ("p") ;
// on supprime la clé "p" par la vue associée
System.out.println ("liste des clés après sup :       " + cles) ;

// modification de la valeur associée à la clé x
String old=m.put("x", "25") ;
if (old != null)
System.out.println ("valeur associée a x avant modif :    " + old) ;
System.out.println ("map après modif de x :         " + m) ;
System.out.println ("liste des valeurs après modif de x : " + valeurs)
;

// On parcourt les entrées (Map.Entry) du map jusqu'a trouver la valeur 20
// et on supprime l'élément correspondant (suppose exister)
Set<Map.Entry<String, String> > entrees=m.entrySet () ;
Iterator<Map.Entry<String, String> > iter=entrees.iterator() ;
while (iter.hasNext())
{ Map.Entry<String, String> entree   =iter.next() ;
String valeur=entree.getValue() ;
if (valeur.equals ("20"))
{ System.out.println ("valeur 20 " + "trouvée en clé "+ entree.getKey()) ;
iter.remove() ; // suppression sur la vue associée
break ;
}}
System.out.println ("map après sup élément dont la valeur est 20 :
                                        " + m) ;
// on supprime l'élément de clé "f"
m.remove ("f") ;
System.out.println ("map après suppression f :    " + m) ;
System.out.println ("liste des clés après suppression f : " + clés) ;
System.out.println ("liste des valeurs après supp de f : " + valeurs) ;
}}
```

L'exécution est la suivante:

```
map initial :                          {p=50, c=10, f=20, g=60, x=40, k=30}
valeur associée a f :                  20
liste des valeurs initiales :          [50, 10, 20, 60, 40, 30]
liste des valeurs après sup :          [50, 10, 20, 60, 40]
liste des clés initiales :             [p, c, f, g, x]
liste des clés après sup :             [c, f, g, x]
valeur associée a x avant modif :      40
map après modif de x :                 {c=10, f=20, g=60, x=25}
liste des valeurs après modif de x :   [10, 20, 60, 25]
valeur 20 trouvée en clé f
map après sup élément de valeur 20 : {c=10, g=60, x=25}
map après suppression f :              {c=10, g=60, x=25}
liste des clés après suppression f : [c, g, x]
liste des valeurs après supp de f : [10, 60, 25]
```

5.3 TreeMap

- **Définition**

La classe **TreeMap** implémente l'interface **SortedMap** et cette dernière hérite de l'interface **Map**.

Cette classe permet de stocker des couples (clé, valeur), dans une structure d'arbre binaire équilibré.

Cette classe garantit que la collection **Map** sera **triée selon un ordre croissant, conformément à l'ordre naturel des clés ou à l'aide d'un comparateur fourni au moment de la création de l'objet TreeMap**. TreeMap ne contient **que des éléments uniques** et ne **peut pas avoir de clé nulle** mais peut avoir plusieurs valeurs nulles.

- **Méthodes les plus utilisées de la classe TreeMap**

Méthode	Description
TreeMap()	Construt une nouvelle arborescence vide en utilisant l'ordre naturel de ses clés.
TreeMap(Comparator<? super K> comparator)	Construit une nouvelle arborescence vide, ordonnée selon le comparateur donné.
TreeMap(Map<?extends K, ? extends V> m)	Construit une nouvelle carte arborescente contenant les mêmes mappages que la

	carte donnée, ordonnée selon l'ordre naturel de ses clés.
int **size**()	Retourne le nombre de couples (clé, valeur) contenus dans ce TreeMap.
Comparator<? super K> **comparator**()	Retourne le Comparator utilisé par le TreeMap, ou null si le TreeMap n'utilise pas de Comparator.
boolean **containsKey**(Object cle)	Retourne true si le TreeMap contient la clé cle.
boolean **containsValue**(Object valeur)	Retourne true si le TreeMap contient la valeur valeur.
V **get**(Object cle)	Retourne la valeur associée à la clé cle.
K firstKey()	Retourne la première clé du TreeSet(la plus petite)
K lastKey()	Retourne la dernière clé du TreeSet(la plus grande)
int size()	Retourne le nombre de couples (clé, valeur) contenus dans ce TreeMap.

Conclusion

Pour conclure ci-joint le tableau récapitulatif des différentes collections

Collection Interface Concrete Implementation Classes

Class	Map	Set	List	Ordered	Sorted
HashMap	x			No	No
Hashtable	x			No	No
TreeMap	x			Sorted	By *natural order* or custom comparison rules
LinkedHashMap	x			By insertion order or last access order	No
HashSet		x		No	No
TreeSet		x		Sorted	By *natural order* or custom comparison rules
LinkedHashSet		x		By insertion order	No
ArrayList			x	By index	No
Vector			x	By index	No
LinkedList			x	By index	No
PriorityQueue				Sorted	By to-do order