Chapitre 2 : Les collections

1. Collections

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 frameWork 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.
Avant le JDK 5.0, les collections pouvaient contenir des éléments d'un type objet quelconques. Exemple on pouvait créer une liste contenant à la fois des éléments de type Integer, String ... De ce fait l'accès à cet élément nécessite le recours à l'opérateur **cast**.

Depuis le JDK 5.0, les collections sont manipulées par le biais de classes génériques. Ainsi tous les éléments de la collection sont de même type.

2. Les génériques

Le principe de la généricité est de faire des classes qui n'acceptent qu'un certain type d'objets.

La définition de la classe générique est :

```
class ClasseGenerique<T>

{

T monAttribut;

public ClasseGenerique(T monattribut)

{...}

public T maMethode()

{....}

....

}
```

Exemple :

```
class Couple<T> {
        private T x, y;

        public Couple(T premier, T second) {
                x = premier;
                y = second;
        }

        public void affiche()
        { System.out.println("Premier: "+ x + " Second : " + y);    }

        public T getX() {return x;      }
```

```
        public void setX(T x) {this.x = x;}

        public T getY() {return y;}

        public void setY(T y) {this.y = y;}
}
```

Utilisation :

```
public static void main(String[] args) {

    Integer v1=10; Integer v2=20;
    Couple<Integer> c1=new Couple<Integer>(v1,v2);

    c1.affiche();
}
```

Exemple de classe générique à plusieurs paramètres

```
public class CoupleM <T,U> {
        private T x;
        private U y;

        public CoupleM(T premier,U second) {
            x = premier;
            y = second;
        }

        public void affiche()
        {
            System.out.println("Premier: "+ x + " Second : " + y);
        }

        public T getX() {
            return x;
        }

        public void setX(T x) {
            this.x = x;
        }

        public U getY() {
            return y;
        }

        public void setY(U y) {
            this.y = y;
        }

        }
```

Utilisation

```
public static void main(String[] args) {

    Integer v1=10; Double v2=2.5;
    Couple M<Integer, Double> c1=new Couple M <Integer,Double> (v1,v2);

    c1.affiche();
}
```

Dans une définition de classe générique on peut imposer que le paramètre de type dérive d'une classe ou qu'il implémente une interface comme suit :

class Couple <T extends Comparable> {...} // T doit implémenter l'interface Comparable.

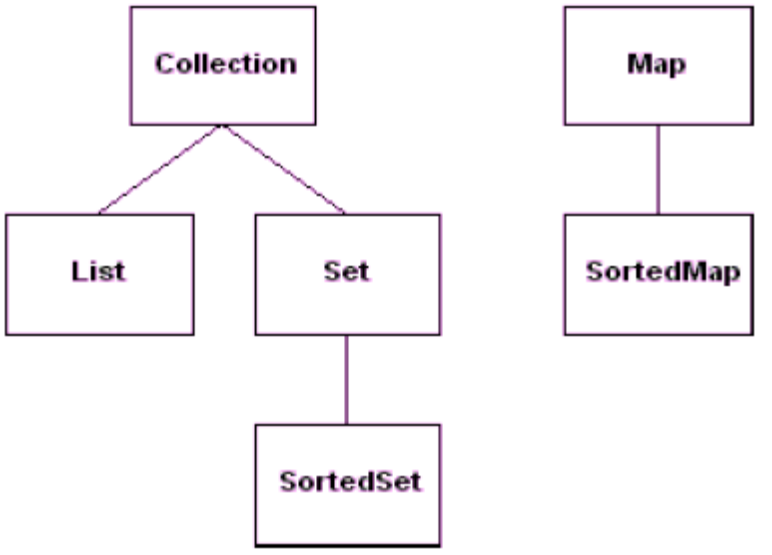
3. Collections et Framework de Java

Le Framewok Java Offre une architecture unifiée pour représenter et manipuler les collections.
Il est composé 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.).

3.1. Interfaces

Organisées en deux catégories: Collection & Map. Les classes et interfaces se trouvent dans le paquetage : java.util.



- **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	

4. Interface Collection

Cette interface offre plusieurs méthodes :

- **size()** et **isEmpty()** : retourne le nombre d'éléments portés par cette collection, et un booléen qui permet de tester si cette collection est vide ou pas.
- **contains(T t)** : retourne true si l'objet passé en paramètre est contenu dans cette collection.
- **add(T t)** et **remove(T t)** : permet d'ajouter (resp. de retirer) un objet à cette collection.
- **iterator()** : retourne un itérateur sur les éléments de cette collection.
- **addAll(Collection<? extends T> collection)** et **removeAll(Collection<? extends T> collection)** : permet d'ajouter (resp. de retirer) l'ensemble des objets passés dans la collection en paramètre.
- **retainAll(Collection<? extends T> collection)** : permet de retirer tous les éléments de la collection qui ne se trouvent pas dans la collection passée en paramètre. Cette opération réalise l'intersection des deux collections.
- **containsAll(Collection<? extends T> collection)** : retourne true si tous les éléments de la collection passée en paramètre se trouvent dans la collection courante. Cette opération teste l'inclusion.
- **clear()** : supprime tous les éléments.

Interface Iterator

C'est une interface permettant d'énumérer les éléments contenus dans une collection.
Toutes les collections proposent une méthode iterator renvoyant un itérateur.
Iterator permet de parcourir dans un sens uniquement.

Le fonctionnement de l'itérateur consiste dans un premier temps, à récupérer un objet **Iterator** par l'intermédiaire de la méthode **iterator()** disponible dans toutes les classes implémentant les interfaces List et Set, telles que ArrayList, LinkedList, HashSet, LinkedHashSet et TreeSet. Puis à l'aide d'une boucle, dont la condition d'arrêt est une expression faisant appel à la méthode **hasNext()** afin de vérifier si l'itérateur possède encore des éléments, récupérer dans un objet, les éléments de l'itérateur en utilisant la méthode **next()**.

Exemple1 :

```
Iterator iterator = collection.iterator();

while (iterator.hasNext()) {

System.out.println("objet = "+iterator.next());}
```

Exemple2 :

```
Iterator iterator = collection.iterator();

if (iterator.hasNext()) {

iterator.next();

iterator.remove();}
```

5. L'interface List

5.1. Présentation

L'interface List modèlise une liste indexée par des entiers. Lorsque l'on ajoute un objet à une liste, il prend un numéro d'ordre, géré par la liste. Lorsque l'on en retire un, il est de la responsabilité de l'implémentation de la liste de conserver une numérotation cohérente.

On peut donc toujours demander le $n^{ième}$ élément d'une liste.

L'interface List étend Collection, et lui ajoute essentiellement deux types de méthodes : celles qui permettent de manipuler les objets directement à partir de leur numéro d'ordre, et celles qui permettent de parcourir la liste dans un sens ou dans l'autre.

Les méthodes que List ajoute à Collection.

- **add(int index, T t)** et **addAll(int index, Collection<? extends T> collection)** : permettent d'insérer un ou plusieurs éléments à la position notée par index.
- **set(int index, T t)** : permet de remplacer l'élément placé à la position index par celui passé en paramètre. L'élément qui existait est retiré de la liste, et retourné par cette méthode.
- **get(int index)** : retourne l'élément placé à l'index passé en paramètre.

- **remove(int index)** : retire l'élément placé à l'index passé en paramètre. Cet élément est retourné par la méthode.
- **indexOf(Object o)** et **lastIndexOf(Object o)** : retournent respectivement le premier et le dernier index de l'objet passé en paramètre dans cette liste.
- **subList(int debut, int fin)** : retourne la liste composé des éléments compris entre l'index debut, et l'index fin - 1. Notons que cette sous-liste n'est pas une copie de la liste existante, mais une vue sur cette liste. Toutes les modifications de la liste principale sont donc vues au travers de cette sous-liste, et réciproquement. Nous verrons cela sur un exemple simple.

5.2. Interface ListIterator

L'interface List supporte bien la méthode iterator(), mais elle ajoute une autre méthode analogue, listIterator(). Cette méthode retourne un objet de type ListIterator, qui est une interface, extension de Iterator.

ListIterator permet de parcourir une liste dans le sens croissant de ses index, ou dans le sens décroissant, et ajoute quelques méthodes supplémentaires.

- **hasPrevious()** : retourne un booléen qui est vrai s'il existe un élément à itérer dans l'ordre décroissant des index.
- **previous()** : retourne l'élément précédent.
- **previousIndex()** et **nextIndex()** : retournent respectivement l'index de l'élément précédent, et l'index de l'élément suivant.

L'interface ListIterator permet aussi de manipuler directement la liste que l'on est en train d'itérer. Ces méthodes sont définies dans l'interface, mais sont facultatives. Une implémentation qui choisirait de ne pas les supporter doit alors jeter une exception de type UnsupportedOperationException lors de leurs appels.

- **add(T t)** : permet d'insérer un élément dans la liste à l'endroit où l'on se trouve, c'est-à-dire avant l'élément qui aurait été retourné par un appel à next(). Notons que si l'on fait un appel à next()après une telle insertion, ce n'est pas l'élément que l'on vient d'insérer qui est retourné, mais l'élément suivant.
- **remove(T t)** : permet de retirer de la liste l'élément que l'on vient d'itérer. Cette méthode ne peut être appelée qu'après un appel à next() ou previous(). Il n'est donc pas légal de faire plusieurs appel de suite à remove(T t). Il est illégal d'appeler cette méthode juste après un appel à add(T t)ou remove(t t).
- **set(T t)** : remplace l'élément que l'on vient d'itérer par l'élément passé en paramètre. Cette méthode ne peut être appelée qu'après un appel à next() ou previous(). Il est illégal d'appeler cette méthode juste après un appel à add(T t) ou remove(t t).

5.3. Implémentations

L'interface List possède trois implémentations: Vector, ArrayList et LinkedList.

ArrayList et LinkedList exposent les mêmes fonctionnalités et la même sémantique. Mais elles ne doivent pas être utilisées dans les mêmes cas. Un tableau permet d'accéder très rapidement à un élément donné si l'on possède son index, ce qui n'est pas le cas d'une liste chaînée. En revanche, l'insertion d'un élément dans un tableau est un processus lourd (il faut

décaler des éléments du tableau). Dans une liste chaînée ce processus est rapide : il s'agit juste d'un mouvement de pointeurs. Enfin, augmenter la capacité d'un tableau est également un processus lourd, alors qu'une liste chaînée, par définition, n'a pas de capacité maximale.

Le type d'implémentation sera donc choisi en fonction du type de problème que l'on a à traiter. D'une façon générale, certaines opérations ensemblistes sont plus rapides sur les listes chaînées, alors que les accès aux éléments individuels sont plus efficaces sur les tableaux.

Exemple :

```
ArrayList<Personne> liste = new ArrayList<Personne>();
liste.add(new Personne("Toto"));
// création d'autres instances de Personne
for (int cpt=0; cpt<liste.size();cpt++)
System.out.println(liste.get(cpt).getNom());
}
```

Exemple : Parcours en utilisant Iterator

```
ArrayList<Personne> liste = new ArrayList<Personne>();
liste.add(new Personne("Toto"));
// création d'autres instances de Personne
Iterator<Personne> it = liste.iterator();
while(it.hasNext()) {
System.out.println(it.next().getNom());
}
```

Parcours en utilisant for

```
ArrayList<Personne> liste = new ArrayList<Personne>();
liste.add(new Personne("Toto"));
// création d'autres instances de Personne
for (Personne p : liste)
System.out.println(p.getNom());
}
```

Exemple d'utilisation d'ArrayList

```
List le = new ArrayList<>();
Employe e = new Employe("Dupond");
le.add(e) ; // Ajoute d'autres employés
 . . .
// Affiche les noms des employés
for (int i = 0; i < le.size(); i++)
 { System.out.println(le.get(i).getNom()); }
```

Exemple : Parcours d'une linkedList à l'envers

```
import java.util.* ;
public class Exemple
{ public static void main (String args[])
{ LinkedList<String> l = new LinkedList<String>() ;
Scanner sc=new Scanner(System.in);
```

```
String continuer="oui";
while (continuer.equals("oui"))
{
                                    System.out.println ("Donnez une suite de mots ") ;
                                    String ch = sc.next() ;
                                    l.add (ch) ;

                                    System.out.println ("Voulez vous continuer ?(oui/non)") ;
                                     continuer = sc.next() ;

}
System.out.println ("Liste des mots :") ;
ListIterator<String> iter = l.listIterator() ;
while (iter.hasNext())
      System.out.print (iter.next() + " ") ;

System.out.println () ;

System.out.println ("Liste des mots à l' envers :") ;
iter = l.listIterator(l.size()) ; // iterateur en fin de liste
while (iter.hasPrevious())
System.out.print (iter.previous() + " ") ;

System.out.println () ;
}
}
```

5.4. LinkedList : méthodes introduites par Java 5 et Java 6

Depuis le JDK 5.0, la classe ***LinkedList*** implémente également l'interface ***Queue***. Depuis Java 6, la classe *LinkedList* implémente en outre l'interface ***Deque*** (queue à double entrée) présentée plus loin.

5.4.1. Interface Queue

L'interface Queue, qui modélise une file d'attente simple, expose six méthodes, qui sont les suivantes.

- **add(T t)** et **offer(T t)** permettent d'ajouter un élément à la liste. Si la capacité maximale de la liste est atteinte, alors add() jette une exception de type IllegalStateException, et offer()retourne false.
- **remove()** et **poll()** retirent toutes les deux de cette file d'attente. Si aucun élément n'est disponible, alors remove() jette une exception de type NoSuchElementException, tandis que poll()retourne null.

- **element()** et **peek()** examinent toutes les deux l'élément disponible, sans le retirer de la file d'attente. Si aucun élément n'est disponible, alors element() jette une exception de typeNoSuchElementException, tandis que peek() retourne null.

5.4.2. Interface Deque

Cette interface est une extension de Queue, ajoutée en Java 6.

Elle définit la notion de file d'attente à double extrémité : il est possible d'ajouter des éléments au début de la file ou à la fin, avec la même sémantique que pour Queue.

Suivant la logique de la construction de Queue, Deque expose trois types de méthodes, qui permettent d'ajouter, de retirer, ou d'examiner l'élément suivant de la file. Chacune de ces méthodes existe en deux versions : la première agit sur une extrémité de la file, et la seconde sur l'autre. Chacune de ces méthodes existe encore en deux versions : la première jette une exception en cas d'échec, la seconde retourne false. Cela fait un total de douze méthodes, que l'on peut regrouper dans les tableaux suivants.

Méthodes de Deque

	génère une exception	**Retourne false**
Insertion	addFirst(T t) / addLast(T t)	offerFirst(T t) / offerLast(T t)
Retrait	removeFirst(T t) / removeLast(T t)	pollFirst(T t) / pollLast(T t)
Examen	getFirst(T t) / getLast(T t)	peekFirst(T t) / peekLast(T t)

L'interface Deque expose également les deux méthodes iterator() et descendingIterator(), qui permettent d'itérer dans les deux sens de la file d'attente.

Enfin, elle expose deux méthodes removeFirst(Object) et removeLast(Object), qui permettent de retirer respectivement la première et la dernière occurrence de l'objet passé en paramètre.

6. Notion de SET

L'interface Set modélise un ensemble d'objets dans lequel on ne peut pas trouver de doublons. Cette notion impose que l'égalité entre objets soit définie, ce qui est le cas, puisque tout objet Java dispose d'une méthode equals().

6.1.Interface SortedSet

6.1.1. 5.1. Notion de SortedSet

L'interface SortedSet est une extension de l'interface Set. Cette interface impose de plus que tous les objets enregistrés dans cet ensemble sont automatiquement triés dans un ordre que nous allons préciser. L'itération sur les éléments d'un SortedSet se fait dans l'ordre croissant associés aux objets de cet ensemble.

La comparaison de deux objets n'est pas définie au niveau de la classe Object. Il y a deux façons de faire pour comparer deux objets.

- Implémenter l'interface Comparable. Cette interface expose une unique méthode : compareTo(T t), qui retourne un entier. Le fait que cet entier soit négatif ou positif nous dit si l'objet comparé est plus grand ou plus petit que notre objet.
- Fournir au SortedSet, lors de sa construction, une instance de Comparator. L'interface Comparatorn'expose une méthode : compare(T t1, T t2), qui a la même sémantique que la méthode compareTo(T t) de Comparable.

6.1.2. 5.2. Détails des méthodes disponibles

L'interface SortedSet propose les méthodes supplémentaires suivantes.

- comparator() : retourne l'objet instance de Comparator qui permet la comparaison, s'il existe.
- first() et last() : retournent le plus petit objet de l'ensemble, et le plus grand, respectivement.
- headSet(T t) : retourne une instance de SortedSet contenant tous les éléments strictement plus petit que l'élément passé en paramètre. Ce sous-ensemble est une vue sur l'ensemble sur lequel il est construit. Il reflète donc les changements de cet ensemble, et réciproquement.
- tailSet(T t) : retourne une instance de SortedSet contenant tous les éléments plus grands ou égaux que l'élément passé en paramètre. Ce sous-ensemble est une vue sur l'ensemble sur lequel il est construit. Il reflète donc les changements de cet ensemble, et réciproquement.
- subSet(T inf, T sup) : retourne une instance de SortedSet contenant tous les éléments plus grands ou égaux que inf, et strictement plus petits que sup. Là encore, ce sous-ensemble est une vue sur l'ensemble sur lequel il est construit, qui reflète donc les changements de cet ensemble, et réciproquement.

6.2.Interface NavigableSet

6.2.1. 6.1. Notion de NavigableSet

L'interface NavigableSet est une extension de SortedSet, introduite en Java 6. La classe d'implémentation fournie en standard est la même que pour SortedSet : il s'agit de TreeSet. Elle ne modifie pas la sémantique définie pour SortedSet, mais ajoute des méthodes et des fonctionnalités.

6.2.2. 6.2. Détails des méthodes disponibles

Cette interface ajoute les méthodes suivantes à SortedSet.

- iterator() et descendingIterator() : retournent deux itérateurs, qui permettent de balayer les objets de cet ensemble dans l'ordre croissant et décroissant, respectivement.
- ceiling(T t) et floor(T t) : retournent le plus petit élément égal ou plus grand que l'élément passé en paramètre, et le plus grand élément égal ou plus petit que celui passé en paramètre, respectivement.
- higher(T t) et lower(T t) : retournent le plus petit élément strictement plus grand que l'élément passé en paramètre, et le plus grand élément strictement plus petit que celui passé en paramètre, respectivement.
- headSet(T sup, boolean inclusive) et tailSet(T inf, boolean inclusive) : retournent unNavigableSet contenant les éléments plus petits (respectivement plus grands) que l'élément passé en paramètre, inclus ou non, suivant la valeur du booléen inclusive. Encore une fois, ce sous-ensemble est une vue de l'ensemble maître.
- subSet(T inf, boolean infInclusive ,T sup, boolean supInclusive) : retourne un NavigableSetcontenant les éléments plus grands que inf et plus petits que sup. L'inclusion ou non des bornes est précisée par les deux booléens passés en paramètre.
- pollFirst() et pollLast() retirent de l'ensemble sa plus petite valeur, ou sa plus grande respectivement, et la retournent.

6.3.Implémentation : TreeSet

- La classe TreeSet<T> implémente les interfaces : Set, SortedSet et NavigeableSet et propose une organisation sous forme d'arbre binaire qui ordonne les objets

Pour ordonner les éléments de la TreeSet, on peut procéder de deux manières :

A- A classe T est comparable c'est à dire elle implémente l'interface Comparable et propose une définition pour la méthode compareTo.

NB: dans TreeSet, equals n'intervient pas du tout

Exemple 1 :
La classe Etudiant implémente l'interface Comparable et de ce fait elle définit la méthode compareTo:

```
public class Etudiant implements Comparable<Etudiant>{…

Ajout de la méthode compareTo

public int compareTo(Etudiant e) {
if (this.getNCE() == null && e.getNCE() == null) return 0;
                if (this.getNCE()==null) return -1;
                return this.getNCE().compareTo(e.getNCE());      }


g = new TreeSet<Etudiant>();
```

```
g.add(new Etudiant("300","MOHAMED",12));
g.add(new Etudiant("200","SALEH",10));
g.add(new Etudiant("400","ALI",14));
System.out.println(g);
```

Résultat :

```
["200","SALEH",10
, "300","MOHAMED",12
, "400","ALI",14]
```

B- Pour ordonner les éléments de la TreeSet on peut utiliser un Comparator qu'on fait passer au constructeur.

Exemple 2 :

```
class ComparateurSelonNom implements Comparator<Etudiant>
    {
      public int compare(Etudiant e1, Etudiant e2) {
            if (e1.getNom() == null && e2.getNom() == null) return 0;
            if (e1.getNom()==null) return -1;
            return e1.getNom().compareTo(e2.getNom());
    }
```

Utilisation

```
    g = new TreeSet<Etudiant>(new ComparateurSelonNom());
    g.add(new Etudiant("300","MOHAMED",12));
    g.add(new Etudiant("200","SALEH",10));
    g.add(new Etudiant("400","ALI",14));
    System.out.println(g);
```

Résultat :

```
    ["400","ALI",14
    ,"300","MOHAMED",12
    ,"200","SALEH",10 ]
```

7. Tables associatives

7.1. Notion de table associative

Une table associative permet de conserver une information associant deux parties nommées clé et 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

Une table associative est une structure de données qui associe des clés à des valeurs. Une telle structure doit au moins exposer deux fonctionnalités :

- une méthode de type put(key, value), qui permet d'associer un objet à une clé et de l'ajouter;
- une méthode de type get(key), qui retourne la valeur qui a été associée à cette clé, ou null s'il n'y en a pas.
- une méthode de type remove(key), qui supprime la clé de cette table, et la valeur qui lui est associée.

7.2.Interface MAP

Cette interface modélise la forme la plus simple d'une table associative. Comme prévu, elle expose les méthodes de base suivantes.

- put(K key, V value) et get(K key) : ces deux méthodes permettent d'associer une clé à une valeur, et de récupérer cette valeur à partir de cette clé, respectivement.
- remove(K key) : permet de supprimer la clé passée en paramètre de cette table, et la valeur associée.
- keySet() : retourne l'ensemble de toutes les clés de cette table. Cet ensemble ne peut pas contenir de doublons, il s'agit d'un Set<K>, donc les éléments sont de type K. Cet ensemble est une vue associative sur les clés de la table associative. Donc les éléments ajoutés à cette table seront vus dans ce Set. Il supporte les méthodes remove() et removeAll(), mais pas les méthodes d'ajout d'éléments. Retirer une clé de cet ensemble retire également la valeur qui lui est associée.
- values() : retourne l'ensemble de toutes les valeurs stockées dans cette table associative. À la différence de l'ensemble des clés, l'ensemble des valeurs peut contenir des doublons. Il est donc de type Collection<V>. Cette collection est également une vue sur la table : toute valeur ajoutée à la table sera vue dans cette collection. Elle supporte les méthode remove() et removeAll(), qui ont pour effet de retirer également la clé associée à cette valeur, mais pas les méthodes d'ajout.
- entrySet() : retourne l'ensemble des entrées de cette table associative. Cet ensemble est un Set, dont les éléments sont de type Map.Entry. Nous allons voir l'interface Map.Entry dans la suite de cette partie. Cet ensemble est lui aussi une vue sur la table, qui reflète donc les modifications qui peuvent y être faites. Il supporte les opérations de retrait d'éléments, mais pas les opérations d'ajout.

Plusieurs autres méthodes utilitaires sont ajoutées à cette interface.

- clear() : efface tout le contenu de la table.
- size() et isEmpty() : retourne le cardinal de la table, et un booléen qui indique si cette table est vide ou pas.
- putAll(Map map) : permet d'ajouter toutes les clés de la table passée en paramètre à la table courante.

- containsKey(K key) et containsValue(V value) : permettent de tester si la clé ou la valeur passée en paramètre sont présentes dans cette table.

Comme on le voit, les méthodes exposées ne sont pas nombreuses, et assez simples à comprendre.

7.3. InterfaceMAP.ENTRY

Cette interface permet de modéliser les couples (clé, valeur) d'une table associative. Elle expose deux méthodes : getKey() et getValue(), qui retournent bien sûr la clé et la valeur de ce couple.

La seule façon d'obtenir un objet de type Map.Entry est de faire un appel à la méthode Map.entrySet(), et d'itérer sur le Set obtenu en retour.

Cet objet est une vue sur la table. Il possède également une méthode setValue(V value), qui permet de modifier la valeur associée à une clé durant une itération.

7.4.Implémentations

– Le J.D.K. propose les classes qui implémentent cette interface :
  - **HashMap** qui stocke les éléments dans une table associative
  - **TreeMap** qui stocke les éléments dans un arbre

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

7.5.Présentation générale de HashMap

Dans le cas de HashMap, on doit définir convenablement

- equals : qui sert à définir l'appartenance d'un élément à l'ensemble

- hashCode : qui est exploitée pour ordonnancer les éléments d'un ensemble, ce qui nous amène à parler de table de hachage.

- la méthode hashCode : **int hascode()**

- Elle doit fournir le code de hachage correspondant à la valeur de l'objet

- Dans la définition de cette fonction, il ne faudra pas oublier que le code de hachage doit être compatible avec equals : deux objets égaux par equals doivent fournir le même code

Exemple :

```
_public int hashCode()
        {       if (getNCE()==null) return 0;
                return getNCE().hashCode();}
```

7.5.1. Exemples d'utilisation : HashMap

- Ajout d'une entrée dans la table

HashMap<String,Integer> m = new HashMap< String,Integer> (); // table vide

m.put("m", new Integer(3)); // ajoute 3 associé à la clé m

- Si la clé fournie à put existe déjà, la valeur associée remplace l'ancienne

- Recherche d'information : get

Integer o = m.get("x"); // fournit la valeur associée à la clé "x"

If(o==null) System.out.println ("aucune valeur associée à la clé "x");

- La méthode **containsKey** permet de savoir si une clé donnée est présente

- Suppression d'information : remove

T cle = "x";

U val =m. remove(cle); // supprime l'élément (clé +valeur) de la clé "x";

if (val !=null)

System.out.println(" on a supprimé l'élément de clé " + cle + " et de valeur " + val);

else System.out.println(" la clé " + cle + " n'existe pas ");

Parcours d'une table associative

- En théorie, HashMap et TreeMap ne disposent pas d'itérateurs, mais à l'aide d'une méthode nommée entrySet, on peut voir une table comme un ensemble de paires

- Une paire est un élément de type Map.Entry

- Les méthodes getKey et getValue permettent de récupérer respectivement la clé et la valeur

Parcours d'une table

- Voici un canevas de parcours d'une table utilisant ces possibilités

```
HashMap<String ,Integer>  m;
...
Set<Map.Entry<String ,Integer>>  entrees = m.entrySet(); //entrees est un ensemble de paires
iterator iter =entrees.iterator(); //itérateur sur les paires
while(iter.hasNext()) //boucle sur les paires
{
Map.Entry<String ,Integer>  entree =(Map.Entry)iter.next; //paire courante
String cle = entree.getKey(); //Clé de la partie courante
Integer valeur = entree.getValue(); //valeur de la paire courante
```