**Les collections**

# Introduction

* **Collections**

Les collections sont des classes qui permettent de manipuler les principales structures de donnés telles que les vecteurs dynamiques (vector et arrayList), les ensembles (HashSet et TreeSet), les listes chaînées (LinkedList) et les tables associatives (Hmap et TreeMap). Ce sont des objets qui servent à stocker d'autres objets.

Les classes relatives à ces structures de données implémentent une même interface (**collection**) qu'elles complètent par leurs propres fonctionnalités.

Elles sont réparties en deux groupes : Interfaces et Implémentations :

**Interfaces**

Les interfaces sont organisées en deux catégories: Collection & Map :

****

* **Collection**: un groupe d'objets manipulés comme s'il s'agit d'un seul objet. Dans une collection la duplication peut être autorisée.
* **Set**: ensemble ne contenant que des valeurs et ces valeurs ne sont pas dupliquées.

Par exemple l'ensemble A = {1,2,4,8}.

* + **Set** hérite de Collection, mais n'autorise pas la duplication.
  + **SortedSet** est un Set trié.
* **List**: hérite 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***.

Ces structures sont capables de contenir des objets de différents types accessibles séquentiellement.

* **Map**: 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** traite des données composées (clé, valeur).

* **SortedMap** est un **Map** trié.



* **Implémentation**
* Il n'y a pas d'implémentation de l'interface **Collection**.
* Pour **Set** et **Map** l'implémentation est soit sous forme d'une table de hachage **HashSet** ou **HashMap** ou bien sous forme d'un arbre **TreeSet** ou **TreeMap**.
* Pour **List** l'implémentation se fait soit sous la forme de tableau (**ArrayList**) ou une liste chaînée (**LinkedList**).

# Description des Interfaces de collections

1. **L'interface *Collection***

L'interface **Collection** représente la racine de la hiérarchie des collections Java.

* **Ordre des éléments d'une collection**

Certaines collections comme les ensembles sont dépourvues d'ordonnancement de leurs éléments. D'autres en revanche, leurs éléments sont ordonnés suivant leurs positions. En plus parfois on a besoin de classer les éléments à partir de leurs valeurs.

**Lorsqu'il est nécessaire de disposer d'un tel ordre sur une collection, on fait appel à deux interfaces Comparable et Comparator.**

* **L'interface comparable possède une seule méthode** public int compareTo(object o).

Cette méthode retourne un entier :

* négatif si l'objet courant est inférieur à l'objet o (**au sens qu'on veut définir**).
* Nul si les deux objets sont égaux
* Positif si l'objet courant est supérieur à l'objet o (**au sens qu'on veut définir**).
* **Utilisation d'un objet comparateur :**

On fournit en argument du constructeur de la classe ou de la méthode qui nécessite une comparaison un objet qu’on nomme un comparateur.

Les comparateurs peuvent être passés comme argument à une méthode de tri (par exemple) dans le but d'appliquer un ordre précis.

Cette interface contient la méthode : public int compare(object o1, object o2)

Cette méthode retourne un entier :

* négatif si l'objet o1 est inférieur à o2 (**au sens qu'on veut définir**).
* Nul si les deux objets sont égaux
* Positif si l'objet o1 est supérieur à o2 (**au sens qu'on veut définir**).

**Egalité d'éléments d'une collection**

Toutes les collections nécessitent de définir l'égalité de deux éléments

Pour certaines structures comme la classe **FILE** et **String** ou les classes enveloppes, la méthode **equals** se base sur la valeur des objets. En revanche pour d'autres, elle est héritée de la classe **Object** et ne compare que les références des objets. Il faut dans ce cas la redéfinir.

1. **Les itérateurs et leurs méthodes (parcours de la collection)**

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

Ils sont de deux sortes, monodirectionnels et bidirectionnels.

* **Itérateurs monodirectionnels : interface Iterator.**

Chaque classe collection dispose d'une **méthode** nommée ***iterator****()* et fournissant un itérateur monodirectionnel (objet d'une classe implémentant l'interface **Iterator**).

A chaque instant, un itérateur indique une position courante qui désigne soit un élément de la collection soit la fin de la collection et dans ce dernier cas la position courante se trouve après le dernier élément.

On peut obtenir l'objet désigné par un itérateur en appelant la méthode *next()* de l'itérateur. Ce qui avance l'itérateur d'une position. Cette méthode fournit un résultat de type **Object**.

La méthode *HasNext()* de l'itérateur permet de savoir si un itérateur est à la fin de la collection.

**Remarque :** toute collection (ordonnée ou non) peut être parcourue par un itérateur.

**Parcours d'une collection par un itérateur**

c étant une collection

Iteraor iter=c.iterator(); //renvoie un objet implémentant l'interface **Iterator** et désignant le

//premier élément de la collection s'il existe.

While(iter.HasNext()){

Object o=iter.next(); //fournit l'élément désigné par l'itérateur et avance l'itérateur à la

//position suivante. En général on convertira o dans le type effectif des éléments.

Utiliser l'objet o

}

* **Méthode *remove* de l'interface Iterator :** Supprime l'objet retourné par la méthode *next*()

**Exemple : suppression de tous les objets vérifiant une condition**

Iteraor iter=c.iterator();

While(iter.HasNext()){

Object o=iter.next();

if (condition) iter.remove();

}

**Remarque**

La méthode *remove()* ne travaille pas directement sur la position courante de l'itérateur.

Alors que la position courante possède toujours une valeur, l'objet courant peut ne pas exister.

***Exemple :***

L'instruction suivante conduit à une exception IllegalStateException.

Iteratot iter=c.iterator();

//iter.next();

iter.remove();

* **Itérateur bidirectionnel :** l'interface **ListIterator.**

Certaines collections comme les listes chaînées et les vecteurs dynamiques peuvent par nature être parcourues dans les deux sens. Elles disposent d'une méthode nommée *listiterator()* et qui fournit un itérateur bidirectionnel.

L'interface **ListIterator** dérive de la classe **Iterator** et dispose des mêmes méthodes *next*(), *HasNext(), previous*(), *HasPrevious*(), *add*(), *remove()* et *set*() qui permet la modification.

**Méthodes previous**() **et HasPrevious**()

**Exemple** : parcours à l'envers d'une liste chainée (*LinkedList*) ;

l étant une liste chainée;

iter=l.listiterator(l.size()); //l étant une liste chaînée, la position courante est la fin de liste.

While(iter.hasPrevious()){

Object o=iter.previous();

}

**Méthode add**

**Exemple** : ajout d'un élément avant le deuxième élément de la liste.

ListIterator it =c.listiterator();

it.next(); //premier élément=élément courant

it.next(); //deuxième élément=élément courant

it.add(elem); // ajoute elem à la position courante (entre le premier et le deuxième element).

**Méthode set**

La méthode *set()* modifie les références des objets.

**Exemple**

Remplacement par null de tous les éléments d'une collection c vérifiant une condition.

ListIterator it =c.listiterator();

While(it.hasNext()) {

Object o=it.next();

if (condition)

it.set(null);

}

ListIterator a été défini spécialement pour parcourir les listes.

public interface ListIterator extends Iterator {

* boolean hasNext();
* Object next();
* boolean hasPrevious();
* Object previous();
* int nextIndex();
* int previousIndex();
* void remove(); // Optional
* void set(Object o); // Optional
* void add(Object o); // Optional

}

1. **Operations communes à toutes les collections**

* **Constructeurs**

Toute collection C :

* Dispose d'un constructeur sans arguments.

C c = new C(); // crée une collection c vide.

* Possède un constructeur ayant un argument de type collection.

C c1 = new C(c); // crée une collection c1 contenant tous les éléments de la collection c.

* **Modifications indépendantes d'un itérateur**

Toute collection dispose de la méthode *add*(element) indépendante de l'itérateur.

Cette méthode ajoute un objet à la fin de la collection pour une liste chainée ou un vecteur, à un emplacement sans importance pour un ensemble.

**boolean *add*(Object o) : ajoute l'objet spécifié au sein de la collection courante**

elle retourne la valeur true lorsque l'ajout a réussi.

Exemple : remplir une liste à partir d'un tableau

String t[]={"java", "vb", "c", "c++"};

C c = new C();

for (int i=0;i<t.lenght;i++) c.add(t[i]);

**boolean *remove*(Object o)**

**boolean contains(Object o) :** retourne *true* si la collection contient l'objet spécifié.

**void clear() :** supprime tous les éléments de la collection.

**boolean equals(Object o) :** teste l'égalité entre l'objet courant et l'objet spécifié.

**Opérations collectives**

Toute collection dispose des opérations suivantes :

* c.addAll(c1) : ajoute tous les éléments de c1 à la collection c.
* c.removeAll(c1); // supprime de la collection c, tous les éléments de c1.
* c.retainAll(c1); //ne conserve dans c que les éléments de c1.
* c.size() //fournit la taille de c
* c.isEmpty() //teste si la collection c est vide ou non
* c.containsAll (c1) // retourne *true* si la collection courante contient tous les éléments de c1.
* **Object[] toArray() :** c.toarray() //crée un tableau **d'objets** à partir de c.
* **Object[] toArray(Object[] a) :** Retourne un tableau contenant tous les éléments de la collection courante.

1. **Les Listes**

Les listes sont des structures capables de contenir des objets de différents types accessibles séquentiellement.

1. [**L'interface List**](http://www.laltruiste.com/coursjava/collection_list.html)

L'interface *List* est une collection ordonnée. Elle permet la duplication des éléments. Elle est renforcée par des méthodes permettant d'ajouter ou de retirer des éléments se trouvant à une position donnée. Elle permet aussi de travailler sur des sous listes.

Elle étend l'interface *Collection* et est implémentée par les classes *ArrayList*, *LinkedList*, et *Vector*.

Les listes peuvent contenir des éléments dupliqués, ainsi que des valeurs *null*.

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

**Les méthodes**

|  |  |
| --- | --- |
| **void add(int index, Object element)** | insère un objet à l'index spécifié, au sein de la liste courante. |
| **boolean add(Object o)** | ajoute l'élément spécifié à la fin de la liste en cours. |
| **boolean addAll(Collection c)** | ajoute tous les éléments de la collection c à la fin de la liste. |
| **boolean addAll(int index, Collection c)** | insère tous les éléments de la collection c, à l'index donné au sein de la liste. |
| **void clear()** | supprime tous les éléments de la liste courante. |
| **boolean contains(Object o)** | retourne *true* si la liste contient l'élément o. |
| **boolean containsAll(Collection c)** | retourne *true* si la liste contient tous les éléments de la collection c. |
| **boolean equals(Object o)** | teste l'égalité entre l'objet de la liste et l'objet spécifié. |
| **Object get(int index)** | retourne l'élément situé à la position spécifiée dans la liste. |
| **int indexOf(Object o)** | retourne la position de la première occurrence de l'objet au sein de la liste courante, ou -1 s'il n'a pas été trouvé. |
| **boolean isEmpty()** | retourne *true* si la liste ne contient aucun élément. |
| **Iterator iterator()** | retourne un itérateur sur les éléments de la liste en cours. |
| **int lastIndexOf(Object o)** | retourne la position de la dernière occurrence de l'objet spécifié au sein de la liste, ou -1 s'il n'a pu être trouvé. |
| **ListIterator listIterator()** | retourne un itérateur de liste sur les éléments de l'objet *List*. |
| **ListIterator listIterator(int index)** | retourne un itérateur de liste sur les éléments de l'objet *List* à partir de la position spécifiée. |
| **Object remove(int index)** | supprime l'élément positionné à l'index spécifié au sein de la liste. |
| **boolean remove(Object o)** | supprime la première occurrence de l'élément spécifié au sein de la liste. |
| **boolean removeAll(Collection c)** | supprime tous les éléments de la collection c, au sein de la liste, s'ils sont présents. |
| **boolean retainAll(Collection c)** | retient seulement les éléments de la liste qui ne sont pas contenus dans la collection c. |
| **Object set(int index, Object element)** | remplace l'élément situé à la position spécifiée dans la liste, par l'objet passé en argument. |
| **int size()** | retourne le nombre d'éléments contenus dans la liste. |
| **List subList(int fromIndex, int toIndex)** | retourne une partie de la liste, délimitée par les indices *fromIndex* inclus et *toIndex* exclus. |
| **Object[] toArray()** | retourne un tableau contenant tous les éléments de la liste courante. |

1. **Classe LinkedList : Listes chaînées**

La classe LinkedList permet de manipuler des listes doublement chaînées pouvant être utilisées comme des piles ou des files d'attente. A chaque élément de la collection, sont associés deux références, celle de l'élément précédent et celle de l'élément suivant.

Une telle collection peut être parcourue par un itérateur bidirectionnel **ListIterator**.

La classe **LinkedList** implémente l'interface **List***.*

Les listes chaînées acceptent tous types d'objets, y compris la valeur ***null***.

Il existe deux constructeurs de la classe **LinkedList** :

**LinkedList liste = new LinkedList();**// par défaut

**LinkedList liste = new LinkedList(collection);** // initialisation avec les éléments d'une collection fournie en argument.

D'autre part la méthode *listiterator()* fournit un itérateur bidirectionnel avec les méthodes *getFirst()* et *getLast()*.

**Ajout d'un élément**

LinkedList l;

…

ListIterator it =l.listiterator();

it.add(elem);

Si la liste est vide l'ajout se fait au début de la liste.

Si hasNext vaut false, l'ajout se fait à la fin de la liste.

La classe **LinkedList** possède quatre autres méthodes spécifiques aux listes : *addFirst(), addLast(), removefirst() et removeLast()*.

**Exemple 1**

import java.util.\*;

public class tabpoint{

public static void main(String [] arg){

personne p;

int i;

LinkedList l =new LinkedList();

personne t[] =new personne [3];

//remplir le tableau par des personne

//copier le tableau dans la liste l

for(int i = 0; i < T.length; i++)

l.add(T[i]);

//afficher la liste

ListIterator iter=l.listIterator();

while(iter.hasNext()){

p=(personne)iter.next();

p.affiche();

}

for(int i = 0; i < l.size(); i++){

p=(personne) l.get(i));

p.affiche();

}

}}

1. **Les tableaux redimensionnables : class ArrayList**

La classe **ArrayList** permet de mettre en œuvre un tableau dont la taille peut varier dynamiquement.

La classe **ArrayList** implémente l'interface **List** qui étend l'interface **Collection**.

La classe **ArrayList** offre des fonctionnalités rapides comparables à celles d'un tableau d'objets.

**ArrayList** accepte n'importe quel type de données, y compris **null**!

**Construction**

**Trois constructeurs.**

* Le premier permet d'instancier un objet vide avec une capacité initiale égale à 10.

**ArrayList liste = new ArrayList();**// par défaut : capacité initiale = 10

* Un autre accepte comme argument une valeur destinée à définir cette capacité initiale.

**ArrayList liste = new ArrayList(100);** // initialisé avec une capacité initiale

* Le dernier initialise la liste avec les éléments d'une collection qui lui aura été spécifiée.

**ArrayList v1=new ArrayList(c);** // initialisé avec une collection

**Ajout d'un élément**

L'ajout d'éléments dans une collection *ArrayList* se réalise de différentes manières.

* **liste.add(objet);** // ajout d'un objet
* **liste.addAll(collection);** // ajout des éléments d'une collection
* **liste.add(i, objet);** // insertion d'un objet à l'indice spécifié
* **liste.addAll(i, collection);** // insertion d'éléments d'une collection à l'index spécifié

**Accès aux éléments**

**Object obj = liste.get(index);** //connaitre la valeur de l'élément de rang index.

**parcourt de la liste :**

for(int i=0;i<v.size();i++){

obj=v.get(i)

}

**Exemple**

Public void affiche(ArrayList v) {

for(int i=0;i<v.size();i++)

System.out.print(v.get(i)+" ");

System.out.println();

}

**boolean resultat = liste.contains(objet);** // vérifie si la liste contient l'élément spécifié

**boolean resultat = liste.containsAll(collection);** // vérifie si la liste contient tous les éléments de la collection spécifiée

Les méthodes *indexOf()* et *lastIndexOf()* permettent de rechercher l'élément fourni, respectivement à partir du début et de la fin de la liste.

**Object element = liste.indexOf(obj);** // recherche la première occurrence de l'objet spécifié

**Object element = liste.lastIndexOf(obj);**//recherche la dernière occurrence de l'objet spécifié

Les méthodes *subList()* et *toArray()* sont capables d'extraire une partie ou la totalité des éléments d'une liste, dans un objet *List* et un tableau respectivement.

**List sousListe = liste.subList(5, 15);** //retourne une liste contenant les éléments de l'intervalle spécifié.

**Object[] tableau = liste.toArray();**// retourne tous les éléments de la liste dans un tableau

**Suppression**

La suppression d'un ou plusieurs éléments d'une liste s'accomplit par plusieurs méthodes :

*remove()*, *removeRange()*, *retainAll()* et *clear()*.

**liste.remove(10);** // supprime l'objet à l'index spécifié

**liste.removeRange(5, 15);** // supprime les objets compris dans l'intervalle spécifié

**liste.retainAll(colection);** // supprime tous les éléments ne faisant pas partie de la collection spécifiée

**liste.clear();**// supprime tous les éléments de la liste

La taille de la liste est accessible à partir de la méthode *size()*.

**int taille = liste.size();**

La capacité peut être augmentée par la méthode *ensureCapacity()* et réadapter par rapport à la taille courante avec *trimToSize()*.

**liste.ensureCapacity(50);** // augmente la capacité de la liste

// réajustement de la capacité à la taille de la liste

int taille = liste.size(); // taille = 33

**liste.trimToSize();** capacité = 33

La méthode *set()* remplace un élément de rang donné de la liste par un objet spécifié et retourne l'élément remplacé.

**Object eltaRemplace = liste.set(10, nouvelObjet);**

**Exemple1**

import java.util.\*;

public class array1{

public static void main(String[] args) {

ArrayList v=new ArrayList();

for(int i=0;i<10;i++)

v.add(new Integer(i));

//v.remove(3);

//v.add(2,new Integer(200));

v.set(5,new Integer(600));

for(int i=0;i<v.size();i++)

System.out.print(v.get(i) + " ");

System.out.println();

}}

1. [**La classe Vector**](http://www.laltruiste.com/coursjava/collection_vector.html) (ancienne)

La classe ***Vector*** permet de créer une collection d'objets qui fonctionne de la même manière qu'un tableau, mais sa capacité peut varier en fonction des besoins.

Un vecteur comporte quatre constructeurs.

* Le premier permet de créer un vecteur initialement vide.

**Vector vecteur = new Vector();**

* Le second permet d'initialiser le vecteur à une certaine taille fournie en argument. La valeur de chacun des éléments est null.

**Vector vecteur = new Vector(50);**

* Le troisième permet d'initialiser le vecteur à une certaine capacité et du pas d'incrémentation de sa capacité suite à un dépassement.

**Vector vecteur = new Vector(100, 10);**

* Enfin, le quatrième permet de construire un vecteur à partir d'une collection passée en argument.

**Vector vecteur = new Vector(collection);**

**Récupérer un élément : Object *elementAt*(int index)**

La consultation des éléments dans une collection Vector s'effectue par l'intermédiaire de diverses méthodes permettant de récupérer un élément précis ou plusieurs.

La méthode Object *elementAt*(int index) retourne l'élément trouvé à l'index spécifié.

String obj = (String)vecteur.*elementAt*(5);

import java.util.\*;

class Vecteur {

public static void main(String args[]) {

ArrayList collection = new ArrayList();

collection.add("lundi");

collection.add("mardi");

collection.add("mercredi");

collection.add("jeudi");

collection.add("vendredi");

collection.add("samedi");

collection.add("dimanche");

**Vector vecteur = new Vector(collection);**

for(int i = 0; i < vecteur.size(); i++)

System.out.println (vecteur.elementAt(i));

System.out.println ("Taille du vecteur : " + vecteur.size());

} }

* int capacite = **vecteur.capacity()**;
* int taille = **vecteur.size()**;
* *setSize(int taille)* détermine une nouvelle taille pour le vecteur : vecteur.setSize(100);
* *ensureCapacity(int capacite)* augmente la capacité du vecteur

**vecteur.ensureCapacity(vecteur.capacity() + 50);**

* *trimToSize()* adapte la capacité à la taille courante du vecteur :vecteur.trimToSize();
* *La méthode void add(int index, Object element) insère un élément à la position* index.

**vecteur.add(3, "Une chaîne");**

* *La méthode boolean add(Object obj) ajoute un élément à la fin du vecteur.*

***boolean reussi = vecteur.add(obj);***

* La méthode *boolean addAll(Collection col)* ajoute une collection à la fin du vecteur.

boolean reussi = **vecteur.addAll(colllectionObjets)**;

* La méthode *boolean addAll(int index, Collection col)* ajoute une collection d'éléments à l'index spécifié dans le vecteur.

**boolean reussi = vecteur.addAll(2, colObjets);**

* La méthode *void addElement(Object obj)* ajoute un objet à la fin du vecteur.

**vecteur.addElement(String[]);**

* La méthode *void insertElementAt(Object obj, int index)* insère un objet à l'index spécifié du vecteur : **vecteur.insertElementAt("Lundi", 1);**
* La méthode Object get(index) permet d'accéder à un élément du vecteur :

**Object o = liste.get(3);**

* **Les méthodes *Object firstElement()* et *Object lastElement()*** retournent respectivement le premier et le dernier élément du vecteur.
* La méthode boolean contains(Object obj) vérifie si l'objet est contenu dans le vecteur.

**boolean reussi = vecteur.contains("mercredi");**

* La méthode boolean containsAll(Collection col) teste si tous les éléments de la collection sont contenus dans le vecteur.
* La méthode *Object remove(int index)* supprime l'objet à l'index spécifié et le retourne.

**Object element = vecteur.remove(5);**

* La méthode *boolean remove(Object obj)* supprime la première occurrence de l'objet trouvée dans le vecteur :

**boolean reussi = vecteur.remove("mardi");**

* La méthode *boolean removeAll(Collection col)* supprime tous les éléments de la collection qui sont contenus dans le vecteur :

**boolean reussi = vecteur.removeAll(liste);**

* La méthode *void removeAllElements()* supprime tous les éléments et remet à zéro la taille du vecteur :

**vecteur.removeAllElements();**

* La méthode *boolean removeElement(Object obj)* supprime la première occurrence de l'élément du vecteur :

**boolean reussi = vecteur.removeElement("mercredi");**

* La méthode *void removeElementAt(int index)* efface l'élément à l'index spécifié.

**vecteur.removeElementAt(4);**

* La méthode *protected void removeRange(int debut, int fin)* supprime les éléments situés dans un intervalle spécifié par les arguments :

**vecteur.removeRange(2, 6);**

* La méthode *void clear()* supprime tous les éléments dans le vecteur :

**vecteur.clear();**

* La méthode *subList(int debut, int fin)* extrait une liste d'éléments du vecteur depuis un index de début et jusqu'à un index de fin exclus.

**List liste = vecteur.subList(1, 5);** //Extraction des éléments de l'indice 1 à 4.

* La méthode *boolean retainAll(Collection col)* retient les éléments de la collection et supprime tous les autres du vecteur: boolean reussi=vecteur.retainAll(vecteur.subList(1, 5));

Les méthodes *int indexOf(Object obj)* et *int lastIndexOf(Object obj)* recherchent respectivement la première et la dernière occurrence d'un objet donné en testant l'égalité entre les objets à l'aide de la méthode *equals()*.

int position = vecteur.indexOf("Samedi");//retourne -1 puisque non-trouvé

int position = vecteur.lastIndexof("samedi");//retourne 5

Les méthodes *int indexOf(Object obj, int index)* et *int lastIndexOf(Object obj, int index)* recherchent respectivement la première et la dernière occurrence d'un objet donné en démarrant à une certaine position et en utilisant la méthode *equals()*.

*nextIndex()* permet de retourner l'index suivant, *previousIndex()retourne le*  précédent.

La méthode *set()* permet de remplacer un élément : **vecteur.set(10, nouvelObjet);**

1. **La classe Stack**

La classe *Stack* est une pile du type dernier entré/premier sorti (LIFO : Last-In-First-Out).

La classe *Stack* est une extension de *Vector*.

Un seul constructeur permet d'instancier la classe *Stack*.

**Stack pile = new Stack();**

La classe *Stack* possède cinq méthodes permettant d'assurer un comportement d'une pile à tout objet créé à partir de cette classe.

Les opérations d'entrée et de sortie de la pile sont exécutées respectivement par les méthodes *push()* et *pop()*.

// récupère l'objet au sommet de la pile

**Object element = pile.pop();**

// ajoute l'objet spécifié au sommet de la pile et le retourne

**Object element = pile.push(objet);**

Une exception *EmptyStackException* est levée si l'objet *Stack* est vide lors de ces opérations.

La méthode *peek()* retourne l'objet situé au sommet de la pile sans le supprimer.

**Object element = pile.peek();**

La position d'un objet, par rapport au sommet de la liste, peut être récupérée par la méthode *search()*. Si l'objet recherché n'est pas dans la liste, alors la méthode retourne *-1*.

**int position = pile.search(Object o);**

La méthode *empty()* teste si la pile ne contient aucun élément.

**boolean resultat = pile.empty();**

1. **Les ensembles**

Les ensembles (*Set*) sont des structures contenant des éléments non dupliqués.

**L'interface *Set***

**L'interface *Set* représente une collection ne contenant aucun élément en double. c.à.d. les ensembles ne doivent contenir que des éléments uniques.**

Les collections *Set* peuvent accepter des objets de n'importe quel type et au plus un et un seul élément *null*.

Un ensemble ne peut pas contenir un élément de type *Set*.

Les implémentations de l'interface *Set* peuvent ajouter des restrictions supplémentaires, telles qu'interdire les éléments *null* ou de certains types.

Deux classes implémentent l'interface **Set**: **TreeSet** et **HashSet**.

La classe **TreeSet** n'implémente pas directement l'interface **Set** mais elle s'appuie sur l'interface **SortedSet**.

Le parcours d'un ensemble s'effectue via un itérateur obtenu par la méthode *iterator()*.

|  |
| --- |
| **Les méthodes de l'interface Set** |
| **boolean add(Object o)** |
| ajoute l'élément au sein de l'objet *Set*, s'il n'est pas déjà présent. |
| **boolean addAll(Collection c)** |
| ajoute tous les éléments de la collection au sein de l'objet *Set*, s'ils ne sont pas déjà présents. |
| **void clear()** |
| supprime tous les éléments de la collection *Set*. |
| **boolean contains(Object o)** |
| retourne *true* si l'élément est contenu dans l'objet *Set*. |
| **boolean containsAll(Collection c)** |
| retourne *true* si l'objet *Set* contient tous les éléments de la collection. |
| **boolean equals(Object o)** |
| teste l'égalité entre la collection *Set* et l'objet spécifié. |
| **boolean isEmpty()** |
| retourne *true* si l'ensemble ne contient aucun élément. |
| **Iterator iterator()** |
| retourne un itérateur sur les éléments de l'ensemble. |
| **boolean remove(Object o)** |
| supprime l'élément spécifié de la collection *Set*, s'il est présent. |
| **boolean removeAll(Collection c)** |
| supprime tous les éléments contenus dans la collection, de l'objet Set, s'ils y sont présents. |
| **boolean retainAll(Collection c)** |
| retient seulement les éléments de l'objet *Set*, qui ne sont pas contenus dans la collection. |
| **int size()** |
| retourne le nombre d'éléments de l'ensemble. |
| **Object[] toArray()** |
| retourne un tableau contenant tous les éléments de l'objet *Set*. |

1. **Classe HashSet**

La classe *HashSet* est une implémentation de l'interface *Set*. Ses éléments sont stockés dans une table de hashage.

Un seul élément *null* est autorisé dans une instance de classe *HashSet*.

Les objets *HashSet* ne garantissent pas l'ordre des éléments durant une itération de l'ensemble, contrairement aux collections *TreeSet*.

**Construction**

Il existe quatre constructeurs pour la création de collection *HashSet*.

Le constructeur par défaut (*HashSet()*) initialise l'instance avec des valeurs par défaut, soit une capacité initiale et un facteur de charge par défaut égaux respectivement à *16* et à *0.75*.

Toutefois, il est possible d'indiquer soit la capacité initiale seule (*HashSet(int ini)*), soit la capacité initiale et le facteur de charge (*HashSet(int ini, float fac)*).

**HashSet ensemble = new HashSet();**// par défaut

**HashSet ensemble = new HashSet(100);** // capacité initiale

**HashSet ensemble = new HashSet(100, 0.70f); // capacité initiale et facteur de charge**

**HashSet ensemble = new HashSet(c);**

La classe *HashSet* comprend principalement des méthodes d'ajout (*add()* et *addAll()*), de suppression (*remove()*, *removeAll()* *retainsAll()* et *clear()*) et de vérification de présence (*isEmpty()*, *contains()* et *containsAll()*).

En outre, les méthodes *toArray()* héritées de l'interface **Set** permettent de renvoyer l'ensemble des éléments contenus dans la collection **Set**, dans un tableau.

HashSet ensemble = new HashSet();

// remplissage de la collection...

Object[] tableau = ensemble.toArray(); // le type d'exécution du tableau retourné est celui du tableau spécifié.

**Exemple1**

import java.util.\*;

public class exemple {

public static void main(String args[]) {

HashSet set = new HashSet();

set.add("CCCCC");

set.add("BBBBB");

set.add("DDDDD");

set.add("BBBBB");

set.add("AAAAA");

Iterator iterator = set.iterator();

while (iterator.hasNext()) {System.out.println(iterator.next());}

} }

Les méthodes equals() et hashCode() ont une importance capitale dans le fonctionnement d'un **HashSet**. Chaque élément ajouté, recherché ou supprimé fait appel à ces méthodes pour calculer le code de hachage des objets et les comparer entre eux.

Par exemple, l'ajout d'une entrée entraîne la vérification de l'existence de la clé spécifiée. Dans un premier temps, un code de hachage est calculé spécifiquement par la méthode statique hash() appelant hashCode().

Puis des tests d'égalité entre les codes de hachage des clés existantes et celui de la clé fournie sont réalisés en parallèle avec un autre test d'égalité sur les objets à l'aide de la méthode equals().

1. **L'interface SortedSet**

L'interface **SortedSet** définit une collection de type ensemble trié. Elle hérite de l'interface **Set** qui, elle-même étend l'interface **Collection**.

Elle est implémentée par une seule classe permettant de créer un ensemble trié, il s'agît de la collection ***TreeSet***.

Tous les éléments insérés dans une collection ***SortedSet*** doivent implémenter l'interface *java.lang.****Comparable*** ou être accepté par l'objet ***Comparator***. En outre, tous les éléments d'une telle collection doivent être mutuellement comparables, dans le cas contraire une exception ***ClassCastException*** sera levée.

|  |
| --- |
| **Les méthodes** |
| **Comparator comparator()** |
| retourne le comparateur associé à l'objet *SortedSet*, ou *null* s'il utilise un ordre naturel de ses éléments. |
| **Object first()** |
| retourne le premier des éléments de l'objet *SortedSet*. |
| **SortedSet headSet(Object toElement)** |
| retourne une partie de l'objet *SortedSet*, où les éléments sont strictement inférieurs à l'objet *toElement*. |
| **Object last()** |
| retourne le dernier élément de la collection *SortedSet*. |
| **SortedSet subSet(Object fromElement, Object toElement)** |
| retourne une partie de l'objet *SortedSet*, où les éléments sont situés entre l'objet *fromElement* inclus et *toElement* exclus. |
| **SortedSet tailSet(Object fromElement)** |
| retourne une partie de l'objet *SortedSet*, où les éléments sont supérieurs ou égaux à l'objet *fromElement*. |

**Les méthodes héritées de set sont :**

*add, addAll, clear, contains, containsAll, equals, hashCode, isEmpty, iterator, remove, removeAll, retainAll, size, toArray*

1. **Classe TreeSet**

**La classe *TreeSet* est une implémentation de l'interface *SortedSet*.**

Cette classe est un arbre qui représente un ensemble trié d'éléments.

L'insertion d'un nouvel élément est plus lente mais le tri est directement effectué.

L'ordre utilisé est celui indiqué par les objets insérés s'ils implémentent l'interface ***Comparable*** pour un ordre de tri naturel ou fournir un objet de type **Comparator** au constructeur de l'objet TreeSet pour définir l'ordre de tri.

**Exemple**

import java.util.\*;

public class TestTreeSet {

public static void main(String args[]) {

Set set = new TreeSet();

set.add("CCCCC");

set.add("DDDDD");

set.add("BBBBB");

set.add("AAAAA");

Iterator iterator = set.iterator();

while (iterator.hasNext()) {System.out.println(iterator.next());}

}}

**Opérations générales sur les ensembles (TreeSet et HashSet)**

Un ensemble est une collection non ordonné d'éléments. En plus, aucun élément ne peut apparaitre plusieurs fois dans un ensemble. Même si un ensemble n'est pas ordonné, il nécessite une certaine organisation d'information.

Deux démarches différentes ont été utilisées, ce qui mène à deux classes :

* **HashSet** qui utilise une technique de hachage.
* **TreeSet** qui utilise un arbre binaire pour ordonner les éléments.

Les deux classes **hashSet** et **TreeSet** disposent de la méthode *iterator*() qui fournit un itérateur monodirectionnel(*Iterator*) permettant de parcourir les différents éléments de la collection.

HashSet e;

…..

Iterator it=e.iterator();

While(it.hasNext()) {

Object o=it.next();

//use of o

}

* **Ajout d'un élément**

On ne peut pas ajouter un élément à une position précise puisque les ensembles ne disposent pas d'itérateurs bidirectionnels.

La seule façon d'ajouter un élément à un ensemble est d'utiliser la méthode *add*() de l'interface collection.

HashSet e;

Object elem;

…..

Boolean existe=e.add(elem);

if(existe) System.out.println(elem+" existe déjà ");

else

System.out.println(elem+" a ete ajouté");

* **Suppression d'un élément**

La méthode *remove*() renvoie true si l'élément a été trouvé (et donc supprimé) et false sinon.

HashSet e;

Object o;

…..

boolean trouve=e.remove(o);

if(trouve) System.out.println(elem+" a été supprimé ");

else

system.out.println(elem+" n'existe pas");

En plus, on peut utiliser la méthode *remove*() de l'itérateur monodirectionnel. Cette méthode permet de supprimer l'élément courant s'il existe (le dernier retourné par *next*()).

TreeSet e;

…..

Iterator it=e.iterator();

It.next(); It.next();

It.remove();

* **Test de l'existence d'un élément**

La méthode *contains()* permet de tester l'existence d'un élément.

* **Opérations par bloc**
* e1.*addAll*(e2); // place dans e1 tous les éléments de e2.
* e1.*removeAll*(e2); // supprime de e1 tous les éléments de e2. Après exécution on obtient le "complémentaire de e2 par rapport à e1" dans e1.
* e1.retainAll(e2); //garde dans e1 ce qui appartient à e2. A près exécution on obtient l'intersection de e1 et e2 dans e1.

**Exemple avec l'interface Comparable**

**La classe point**

**public** **class** point **implements** Comparable{

**private** **int** x ;

**private** **int** y ;

**public** point( **int** x , **int** y ){

**this**.x = x ;

**this**.y = y ;

}

**public** **void** affiche(){

System.*out*.println("x= "+ x + " et y= "+y ) ;

}

**public** **int** compareTo(Object o){

point p = (point)o;

**if**(**this**.y<p.y)

**return** -1;

**else**

**if**(**this**.y>p.y)

**return** 1;

**return** 0;

}}

public class exemple\_test {

public static void main(String[] args) {

point p;

int i;

TreeSet ts=new TreeSet();

point t[] =new point [3];

t[0]=new point(121,312);

t[1]=new point(10,276);

t[2]=new point(100,100);

for(i=0;i<t.length;i++)

ts.add(t[i]);

Iterator it=ts.iterator();

while(it.hasNext()){

p=(point)it.next();

p.affiche();

}}}

**Exemple avec l'interface Comparator**

**//définir la classe Comparator**

**class** comparateur **implements** Comparator {

**public** **int** compare(Object o1, Object o2) {

point p1 = (point)o1;

point p2 = (point)o2;

**if**(**p1**.y<p2.y)

**return** -1;

**else**

**if**(**p1**y>p2.y)

**return** 1;

**return** 0;

}}

**Changement dans main :**

Comparator c = **new** comparateur();

TreeSet s=**new** TreeSet(c);

…

1. **Les collections sous forme clé/valeur**

Ce type de collection gère les éléments avec deux entités : une clé et une valeur associée. La clé doit être unique et donc il ne peut y avoir de doublons. En revanche la même valeur peut être associée à plusieurs clés différentes.

Avant l'apparition du framework **collection**, la classe dédiée à cette gestion était la classe **Hashtable**.

1. **L'interface Map**

Cette interface est l'une des deux racines de l'arborescence des collections. Les collections qui implémentent cette interface ne peuvent contenir des doublons.

Les objets **Map** tels que **AbstractMap, HashMap, Hashtable, IdentityHashMap, TreeMap** et **WeakHashMap,** dépendent de deux méthodes particulièrement importantes pour le fonctionnement de ces collections, *hashCode()* et *equals()* qui permettent respectivement de calculer le code de hachage des clés, et de tester l'égalité entre deux objets clés.

Lors de la récupération d'une valeur par la méthode *get()*, de l'ajout d'une entrée par *put()* ou de la suppression d'une paire clé/valeur par *remove()*, les méthodes *equals()* et *hashCode()* y jouent un rôle fondamentale, car elles sont chargées d'assurer la cohérence de la collection.

Par exemple, si une nouvelle entrée doit être ajoutée à la collection, et que la clé y est déjà présente, il faut remplacer la valeur de cette clé et non rajouter une nouvelle paire clé/valeur.

|  |
| --- |
| **Les classes imbriquées** |
| **static interface Map.Entry** |
| représente une entrée (clé -> valeur) d'une collection *Map*. |

|  |
| --- |
| **Les méthodes de la classe Map** |
| **void clear()** |
| supprime toutes les paires clé/valeur de la collection. |
| **boolean containsKey(Object key)** |
| retourne *true* si la collection contient une paire clé/valeur correspondant à la clé spécifiée. |
| **boolean containsValue(Object value)** |
| retourne *true* si la collection contient une ou plusieurs clés pointant la valeur spécifiée. |
| **Set entrySet()** |
| retourne un objet *Set* contenant toutes les entrées de la collection. |
| **boolean equals(Object o)** |
| teste l'égalité entre l'objet *Map* et un autre objet. |
| **Object get(Object key)** |
| reetourne la valeur correspondant à la clé spécifiée. |
| **int hashCode()** |
| retourne le code de hachage. |
| **boolean isEmpty()** |
| retourne *true* si la collection ne contient aucune entrée. |
| **Set keySet()** |
| retourne un objet *Set* contenant les clés de l'objet *Map*. |
| **Object put(Object key, Object value)** |
| associe la valeur spécifiée à la clé donnée au sein de la collection. |
| **void putAll(Map t)** |
| copie toutes les paires clé/valeur de la collection spécifiée au sein de l'objet *Map*. |
| **Object remove(Object key)** |
| supprime la paire clé/valeur correspondant à la clé spécifiée. |
| **int size()** |
| retourne le nombre d'entrées dans la collection. |
| **Collection values()** |
| retourne un objet *Collection* contenant les valeurs de l'objet *Map*. |

Deux classes implémentent l'interface **SortedMap** :

* **HashMap** : stocke les éléments dans une table de hachage
* **TreeMap** : stocke les éléments dans un arbre

1. **L'interface SortedMap**

Elle définit une collection de type **Map** triée sur la clé. Elle hérite de l'interface **Map**.

Le tri peut être assuré de deux façons :

* les clés contenues dans la collection implémentent l'interface **Comparable** pour définir leur ordre naturel
* fournir au constructeur de la collection un objet **Comparator** qui définit l'ordre de tri à utiliser.

Elle définit plusieurs méthodes pour tirer parti de cet ordre.

Cet ordre est reproduit lors de l'itération sur les vues de la collection *SortedMap* retourné par les méthodes *entrySet()*, *keySet()* et *values()*.

L'interface **SortedMap** est implémentée par une classe permettant de créer une collection associative triée sur ses clés, il s'agît de la collection *TreeMap*.

Toutes les clés d'une collection **SortedMap** doivent être mutuellement comparables (implémenter l'interface *java.lang.****Comparable*** ou être acceptées par l'objet ***Comparator***), dans le cas contraire une exception ***ClassCastException*** sera levée.

**boolean res = cle1.compareTo(cle2);**

// ou

**boolean res = comparateur.compare(cle1, cle2));**

La classe imbriquée *Map.Entry* représente une entrée (une paire clé/valeur) dans une collection *SortedMap*.

|  |
| --- |
| **Les classes imbriquées héritées de la classe *java.util.Map*** |
| **Map.Entry** |

|  |
| --- |
| **Les méthodes** |
| **Comparator comparator()** |
| retourne le comparateur associé à l'objet *SortedMap* ou *null* s'il utilise un ordre naturel de ses clés. |
| **Object firstKey()** |
| retourne la première clé de l'objet *SortedMap*. |
| **SortedMap headMap(Object toKey)** |
| retourne une partie des entrées, dont les clés sont strictement inférieures à la clé spécifiée, de l'objet *SortedMap*. |
| **Object lastKey()** |
| retourne la dernière clé de l'objet *SortedMap*. |
| **SortedMap subMap(Object fromKey, Object toKey)** |
| retourne une partie des entrées de l'objet *SortedMap*, dont les clés sont situées entre les objets *fromKey* inclus et *toKey* exclus. |
| **SortedMap tailMap(Object fromKey)** |
| retourne une partie des entrées de l'objet *SortedMap*, dont les clés sont plus grandes ou égales à *fromKey*. |
|  |
| **Les méthodes héritées de l'interface *java.util.Map*** |
| **clear, containsKey, containsValue, entrySet, equals, get, hashCode, isEmpty, keySet, put, putAll, remove, size, values** |

1. **La classe TreeMap**

Cette classe gère une collection d'objets sous la forme clé/valeur stockés dans un arbre. Elle implémente l'interface **SortedMap**.

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**.

Il existe quatre constructeurs permettant d'instancier la classe **TreeMap**.

* Le constructeur par défaut (*TreeMap()*) : initialise l'objet en fondant son mode de tri sur l'ordre naturel des éléments.

**TreeMap associations = new TreeMap();**// par défaut

* Un autre constructeur permet de spécifier un comparateur afin d'effectuer le tri des paires clé/valeur de l'objet *TreeMap*.

// comparateur

Comparator **comparateur** = new Comparateur();

**TreeMap associations = new TreeMap(comparateur);**

**Exemple**

import java.util.\*;

public class TestTreeMap {

public static void main(String[] args) {

TreeMap arbre = new TreeMap();

arbre.put(new Integer(3), "zzzzzz");

arbre.put(new Integer(1), "eeeeeeeeeee");

arbre.put(new Integer(2), "aaaaaaaaaaa");

System.out.print(arbre);

Set cles = arbre.keySet();

Iterator iterator = cles.iterator();

while (iterator.hasNext()) {

System.out.println(arbre.get(iterator.next()));

}

System.out.println(arbre);

}}

Set entrees=arbre.entrySet();

Iterator iterator = entrees.iterator();

while (iterator.hasNext()) {

Map.entry entree =(Map.entry)iterator.next();

Object cle=entree.getKey();

Object val=entree.getValue();

System.out.println(cle+val));

}

}

L'objet *Comparator* doit être créé par le programmeur. Pour cela, il est nécessaire d'écrire une classe devant implémenter l'interface *Comparator.*

* L'interface *Comparator* possède deux méthodes *compare()* et *equals()* qui doivent être définies dans cette classe.
* Un objet *TreeMap* défini sans comparateur, utilise les méthodes *compareTo()* et *equals()* des objets contenus dans cet objet *Map*.

public **interface Comparator** {

public int **compare(**Object o1, Object o2**)**;

public boolean **equals(**Object o**)**;

}

Les deux autres constructeurs acceptent comme argument un objet de type **Map** ou **SortedMap**, destinée à initialiser la nouvelle instance. Chaque paire clé/valeur des collections spécifiées sera copiée au sein de l'objet *TreeMap*.

Dans le premier cas, la collection sera triée selon l'ordre naturel de ses clés,

Dans le second cas conformément à l'ordonnancement de l'objet **SortedMap** indiqué.

Map **collection** = new HashMap();

collection.put(cle, valeur);

......

**TreeMap associations = new TreeMap(collection);** // initialisation avec un objet Map

SortedMap collection = new TreeMap(comparateur);

collection.put(cle, valeur);

………….

TreeMap associations = new TreeMap(collection); // initialisation avec un objet SortedMap

La classe **TreeMap** contient des méthodes d'ajout (*put()* et *putAll()*), de suppression (*remove()* et *clear()*), de vérification de présence (*isEmpty()*, *containsKey()* et *containsValue()*), d'information (*size()*) et d'extraction (*get()*).

La classe *TreeMap* fournit deux méthodes *firstKey()* et *lastKey()*, chargées respectivement de récupérer dans un objet *Map* la première clé, soit la plus petite, et la dernière clé, soit la plus grande. La comparaison entre les clés est assurée par l'ordre naturel ou par le comparateur fourni lors de l'instanciation de l'objet.

**Object premier = associations.fisrtKey();**

**Object dernier = associations.lastKey();**

En outre, la collection *SortedMap* comporte plusieurs méthodes capables d'extraire la totalité ou une partie des entrées, ou seulement des clés ou des valeurs stockées dans l'objet *TreeMap*. Les méthodes *headMap()*, *tailMap()*, *subMap()*, *keySet()* et *values()* permettent toutes d'obtenir une vue spécifique de la collection *TreeMap*.

TreeMap associations = new TreeMap();

// remplissage de la collection...

**Set cles = associations.keySet();**// Toutes les clés sont stockées dans la collection cles

**Collection valeurs = associations.values();**//les valeurs sont stockées dans l'objet valeurs.

Les méthodes *headMap()* et *tailMap()* retournent seulement les entrées de l'objet *TreeMap*, dont les clés sont respectivement, strictement inférieures et supérieures ou égales à l'objet passé en argument.

La méthode *subMap()* extrait un intervalle d'entrées, dont les clés sont comprises entre la valeur du premier argument incluse et la valeur du second argument exclue.

**SortedMap partieInferieure = associations.headMap(valeur);**

**SortedMap partieSuperieure = associations.tailMap(valeur);**

**SortedMap partie = associations.subMap(valInf, valSup);**

1. **La classe HashMap**

**La classe HashMap est une implémentation de l'interface Map*.***

Une seule clé *null* est autorisée dans une instance de classe *HashMap*. Par contre, les valeurs *null* peuvent être plusieurs, l'unicité des entrées étant assurées essentiellement par les clés.

Les objets **HashMap** ne garantissent pas l'ordre des éléments durant une itération de l'association, contrairement aux collections **TreeMap**. En effet, la *fonction de hachage* disperse les éléments au sein d'une collection *Map*.

**Constructeurs**

**HashMap** associations **= new HashMap();**

**TreeMap** collection = new **TreeMap**();

collection.put(cle, valeur);

**HashMap associations = new HashMap(collection);**

La classe **HashMap** comprend deux méthodes d'ajout *put()* et *putAll()* permettant respectivement d'ajouter une seule paire clé/valeur et toutes les entrées d'une collection *Map*.

Les méthodes d'ajout vérifient au préalable si les clés spécifiées ne sont pas déjà présentes dans la collection. Si c'est le cas, les valeurs correspondantes aux clés existantes sont remplacées par les nouvelles valeurs spécifiées.

Object entree = associations.put(cle, valeur);

Map collection = new TreeMap();

associations.putAll(collection); // Ajout d'entrées à la collection...

**Object entreeSupprimee = associations.remove(cle);** // supprime l'entrée correspondant à la clé passé en argument

associations.clear();// suppression de toutes les entrées de la collection HashMap

La méthode *get()* permet l'extraction d'une valeur en fonction d'une clé indiquée. L'objet retourné est du type référence *Object*, il sera alors nécessaire dans la plupart des cas de le convertir.

HashMap associations = new HashMap();

associations.put("Une clé", new StringBuffer("Une valeur");

Object obj = associations.get("Une clé");// extraction d'une valeur

La vérification de présence d'entrées, d'une clé ou d'une valeur est possible par des méthodes *isEmpty()*, *containsKey()* et *containsValue()*.

if(**associations.isEmpty()**)

System.out.println("La collection est vide !");

if(**associations.containsKey(cle)**)

System.out.println("La clé 'cle' a été trouvée !");

if(**associations.containsValue(valeur)**)

System.out.println("La valeur 'valeur' a été trouvée !");

La taille de l'objet *HashMap* (le nombre d'entrées contenues) est retourné par *size()*.

**int taille = associations.size();**

En plus, la collection *HashMap* comporte plusieurs méthodes capables d'extraire la totalité des entrées, ou uniquement les clés ou les valeurs stockées dans l'objet *TreeMap*.

HashMap associations = new HashMap();

// remplissage de la collection...

**Set cles = associations.keySet();**// Toutes les clés sont stockées dans la collection **Set**

**Collection valeurs = associations.values();**//les valeurs sont stockées dans l'objet Collection valeurs.

La méthode *entrySet()* retourne toutes les entrées de l'objet *HashMap* au sein d'un ensemble *Set*. Le contenu de cette collection *Set* est un ensemble d'objets *Map.Entry* contenant la clé et la valeur de chaque paire clé/valeur de l'objet *HashMap*.

**Set entrees = associations.entrySet();**

Iterator iterateur = **entrees**.iterator();

while(iterateur.hasNext()){

**Map.Entry entree = (Map.Entry)iterateur.next();**

System.out.println(**entree.getKey()** + " " + **entree.getValue()**);

}

Les méthodes *equals()* et *hashCode()* ont une importance capitale dans le fonctionnement d'un objet *HashMap*. Chaque entrée ajoutée, recherchée ou supprimée fait appel à ces méthodes pour calculer le code de hachage des objets clés et les comparer entre eux.

1. **Les classes utilitaires**

**Les collections Java nécessitent le support de plusieurs classes et interfaces utilitaires pour fonctionner ou exécuter certaines opérations comme les comparaisons.**

Les classes et interfaces utiles sont :

***Collections*** pour l'ensemble des collections,

***Enumeration*** pour les ensembles et les listes,

***Map.Entry*** pour les maps, etc..

1. [**La classe Collections**](http://www.laltruiste.com/coursjava/collection_collections.html)

**La classe *Collections* est une classe utilitaire pour les collections Java. Cette classe contient des méthodes statiques qui manipulent ou retournent des collections.**

Toutes les méthodes de la classe *Collections* sont susceptibles de lever une exception *NullPointerException* si les collections passées en argument possèdent la valeur *null*.

Trois champs statiques et constants sont contenus dans la classe *Collections*, pour représenter une collection vide.

**List liste = Collections.EMPTY\_LIST;**

**Set ensemble = Collections.EMPTY\_SET;**

**Set associations = Collections.EMPTY\_MAP;**

La méthode *reverseOrder()* retourne un objet *Comparator* qui impose un tri inverse à celui de l'ordre naturel d'une collection.

List **liste** = new LinkedList();

liste.add("Un élément");

// ...

liste.add("Dix éléments");

**int res = Collections.sort(liste, Collections.reverseOrder());**

Les méthodes *min()* et *max()* retournent respectivement l'élément minimum ou maximum d'une collection spécifiée, selon l'ordre naturel de ses éléments ou d'un comparateur fourni. Un objet de type Object est retourné par ces méthodes.

**Object oMin = Collections.min(tableau);**

**Object oMax = Collections.max(tableau);**

Plusieurs méthodes permettent de modifier l'ordre initial d'une collection *List*. La méthode *swap()* permet d'échanger la position d'éléments en indiquant leurs index.

**Collections.swap(liste, 5, 8);**

La méthode *rotate()* fait glisser dans un sens ou dans l'autre, l'élément pointé dans une sorte de cercle joignant les deux extrémités de la liste.

List **liste** = new LinkedList();

for (int i = 0; i <= 10; i++)

liste.add(new Integer(i));

Collections.rotate(liste, 1); // sens avant

// liste[10, 0, 1, ..., 9]

Collections.rotate(liste, -1); // sens arrière

// liste[1, 2, ..., 10, 0]

Les méthodes *shuffle()* permutent aléatoirement la liste spécifiée en utilisant un algorithme aléatoire par défaut ou précisé par un objet *Random*.

**Collections.shuffle(liste);**

**Collections.shuffle(liste, objRandom);**

L'inversion complète de l'ordre des éléments d'une liste peut être obtenue par la méthode *reverse()*.

List **liste** = new LinkedList();

for (int i = 0; i <= 10; i++)

liste.add(new Integer(i));

**Collections.reverse(liste);**

// liste[10, 9, ..., 0]

La méthode *fill()* remplace tous les éléments de la liste indiquée par l'objet passé en argument.

**Collections.fill(liste, null);**

Il est possible de rechercher une liste cible au sein d'une liste source par l'intermédiaire des méthodes *indexOfSubList()* et *lastIndexOfSubList()*. La position de la première occurrence ou de la dernière de la liste cible, à l'intérieur de la liste source, est retournée, sinon, -1 est renvoyée.

List **listeSource** = new LinkedList();

for (int i = 0; i <= 10; i++)

listeSource.add(new Integer(i));

List **listeCible** = new LinkedList();

for (int i = 3; i <= 7; i++)

listeCible.add(new Integer(i));

**Collections.indexOfSubList(listeSource, listeCible);**

// retourne 3

**Collections.lastIndexOfSubList(listeSource, listeCible);**

// retourne 7

La méthode *copy()* permet de copier une liste de la classe *Collections*. Elle recopie tous les éléments d'une liste source vers une liste cible.

**Collections.copy(listeSource, listeCible);**

La méthode *nCopies()* a pour fonction de créer une liste qui sera composée d'un certain nombre d'occurrences de l'objet spécifié.

**List liste = Collections.nCopies(100, "bonjour");**

La méthode *replaceAll()* permet de substituer un élément existant par un nouvel objet, au sein d'une liste.

boolean res = Collections.replaceAll(liste, ancValeur, nouvValeur);

Les singletons, ou ensembles constitués d'un seul élément, sont réalisables pour chaque genre de collections Java, par le biais des méthodes *singleton()* (pour les *Set*), *singletonList()* et *singletonMap()* pour les autres. Chacune de ces méthodes prend en argument un élément pour les collections *Set* et *List* ou une entrée pour les *Map*.

**Set ensemble = Collections.singleton(objet);**

**List liste = Collections.singletonList(objet);**

**Map associations = Collections.singletonMap(cle, valeur);**

La possibilité de fournir des collections accessibles en lecture seule non-modifiables, est donnée par les méthodes particulières préfixées par *unmodifiable*.

**Collection collection = Collections.unmodifiableCollection(collection\_Source);**

**List liste = Collections.unmodifiableList(listeSource);**

**Map associations = Collections.unmodifiableMap(mapSource).**

**Set ensemble = Collections.unmodifiableSet(ensSource);**

**SortedMap mapTrie = Collections.unmodifiableSortedMap(sortedMapSource);**

**SortedSet setTrie = Collections.unmodifiableSortedSet(sortedSetSource);**

1. [**Les interfaces Iterator**](http://www.laltruiste.com/coursjava/collection_iterator.html) **et ListIterator (déjà vue).**
2. [**L'interface java.lang.Comparable**](http://www.laltruiste.com/coursjava/collection_comparable.html)

**L'interface *Comparable* impose un ordre global sur les instances des classes implémentant cette interface.**

L'interface *Comparable* contient une seule méthode. Il s'agît de la méthode *compareTo()* ayant pour fonction de comparer un objet passé en argument à l'instance de classe courante.

**public** **interface Comparable** {

public int **compareTo(**Object o**)**;

}

L'ordre imposé par cette interface se réfère à l'ordre naturel de la classe et la méthode *compareTo()* de la classe se réfère également à sa méthode de comparaison naturelle.

Les listes et les tableaux d'objets qui implémentent l'interface *Comparable* peuvent être respectivement triés par les méthodes *Collections.sort()* et *Arrays.sort()*.

Si l'objet courant ne peut être comparé à l'objet spécifié, la méthode *compareTo()* lancera une exception ***ClassCastException***.

La valeur *null* n'étant pas une instance de classe, une exception ***NullPointerException*** sera levée si une tentative de comparaison avec *null* est tentée.

**boolean res = objet.compareTo(null); // provoque une exception**

L'implémenteur doit s'assurer des règles suivantes :

* sgn(x.compareTo(y)) == -sgn(y.compareTo(x))
* (x.compareTo(y) > 0 && y.compareTo(z) > 0) implique que x.compareTo(z) > 0
* x.compareTo(y) == 0 🡺 sgn(x.compareTo(z)) == sgn(y.compare(z)) pour tout z
* (x.compareTo(y) == 0) <==> (x.equals(y))

**Exemple avec l'interface Comparable**

**La classe point**

**public** **class** point **implements** Comparable{

**private** **int** x ;

**private** **int** y ;

**public** point( **int** x , **int** y ){

**this**.x = x ;

**this**.y = y ;

}

**public** **void** affiche(){

System.*out*.println("x= "+ x + " et y= "+y ) ;

}

**public** **int** compareTo(Object o){

point p = (point)o;

**if**(**this**.y<p.y)

**return** -1;

**else**

**if**(**this**.y>p.y)

**return** 1;

**else**

**return** 0;

}}

public class main1 {

public static void main(String[] args) {

point p;

int i;

TreeSet l=new TreeSet();

point t[] =new point [3];

t[0]=new point(121,312);

t[1]=new point(10,276);

t[2]=new point(100,100);

for(i=0;i<t.length;i++)

l.add(t[i]);

Iterator it=l.iterator();

while(it.hasNext())

{

p=(point)it.next();

p.affiche();

}}}

**Exemple avec l'interface Comparator**

**Changement dans main :**

Comparator c = **new** comparateur();

TreeSet s=**new** TreeSet(c);

**//Classe Comparator**

**class** comparateur **implements** Comparator {

**public** **int** compare(Object o1, Object o2) {

point p1 = (point) o1;

**return** p1.compareTo(o2);

}

1. [**L'interface Comparator**](http://www.laltruiste.com/coursjava/collection_comparator.html)

**L'interface *Comparator* permet de réaliser une comparaison qui impose un ordre global sur les éléments d'une collection.**

Les comparateurs peuvent être passés comme argument à une méthode de tri dans le but d'appliquer un contrôle précis sur l'ordre de tri. Ils peuvent aussi être utilisés pour contrôler l'ordre de certaines structures de données, telles que les objets *TreeSet* et *TreeMap*.

Comparator comparateur = new Comparator() {

public int compare(Object o1, Object o2){

// instructions...

**}**

**};**

L'ordre imposé par un objet *Comparator* est dit être consistant avec *equals()* si et seulement si *(compare((Object)e1, (Object)e2)==0)* retourne la même valeur booléenne que l'expression *e1.equals((Object)e2)* pour chaque *e1* et *e2* dans l'ensemble.

**Les méthodes**

|  |  |
| --- | --- |
| int compare(Object o1, Object o2) | compare les deux objets pour indiquer leur ordre. |
| boolean equals(Object obj) | indique si l'objet spécifié est égal à l'implémentation de l'interface Comparator. |

La méthode *compare()* retourne un nombre entier indiquant l'ordre des objets soumis.

int res = comparateur.compare(arg1, arg2);

* Un entier négatif signifie que le premier argument est inférieur au second.
* Une valeur égale à zéro signifie que le premier argument est égal au second.
* Un entier positif signifie que le premier argument est supérieur au second.

1. [**L'interface Map.Entry**](http://www.laltruiste.com/coursjava/collection_mapentry.html)

**L'interface statique *Map.Entry* est imbriquée dans l'interface *Map*. Elle représente une entrée, c'est-à-dire une paire clé/valeur, d'un objet *Map*.**

La méthode *Map.entrySet()* retourne un objet *Set*, contenant les paires clé/valeur de la collection *Map* sous la forme d'objets *Map.Entry*.

|  |  |
| --- | --- |
| **Les méthodes** |  |
| **boolean equals(Object o)** | teste l'égalité entre l'objet spécifié et l'objet *Map.Entry*. |
| **Object getValue()** | retourne la valeur correspondant à l'entrée en cours. |
| **Object getKey()** | retourne la clé correspondant à l'entrée en cours. |
| **int hashCode()** | retourne le code de hachage pour l'entrée en cours. |
| **Object setValue(Object value)** | fixe la valeur de l'entrée en cours avec l'objet spécifié. |

1. [**La classe Arrays**](http://www.laltruiste.com/coursjava/collection_arrays.html)

**La classe *Arrays* contient diverses méthodes destinées à manipuler les tableaux Java.**

La classe *Arrays* est statique et donc ne peut être instanciée. Elle contient essentiellement des méthodes statiques.

**Type valeur = Arrays.methodeStatique(tableau, autreArguments);**

Les méthodes permettent principalement d'effectuer des recherches et le tri dans un tableau.

int[ ] tableau = {8, 1, 6, 10, 9, 3, 7};

**Arrays.sort(tableau);**

**Arrays.sort(tableau, 2, 5);** // tri de cinq éléments du tableau à partir du rang 2, soit {6, 1, 10, 9, 3}

En ce qui concerne les objets, le tri peut être réalisé au moyen de deux manières.

* La première applique un tri en accord avec l'ordre naturel des éléments. Tous les éléments du tableau doivent impérativement implémenter l'interface Comparable et doivent être mutuellement comparable, c'est-à-dire, qu'une exception ClassCastException ne doit pas être levée lors de l'appel de la méthode compareTo().

boolean res = elt1.compareTo(elt2);

* Dans le second cas, un comparateur est passé en argument avec le tableau, afin d'indiquer à la méthode *sort()* le mode de tri à mettre en œuvre sur ce tableau.

Le comparateur spécifié inclus un algorithme de comparaison spécifique au type référence des objets du tableau.

**Exemple**

class comparateur implements Comparator {

public int compare(Object o1, Object o2) {

point p1 = (point) o1;

point p2 = (point) o2;

comparer p1 et p2

return valeur

} }

Prise en compte du **compareTo** uniquement. Nous allons considérer le tri suivant l'ordre naturel.

Collections.sort(liste);

Prise en compte du compare uniquement. Nous allons considérer le tri suivant le comparateur.

Collections.sort(liste, comparateur)

Il est possible aussi de spécifier un intervalle d'éléments à trier, au sein du tableau d'objets.

**Arrays.sort(tabObjets, 3, 6, comparateur);**

**Méthodes de recherche : binarySearch()**

Les méthodes *binarySearch()* sont utilisées pour accomplir une recherche d'un élément spécifié au sein d'un tableau trié par la méthode *sort()*.

En cas de réussite de la recherche, l'index de l'élément dans le tableau est retourné, sinon *-1* est retourné.

char[] tableau = {'a', 'r', 'k', 'q', 'o', 's'};

Arrays.sort(tableau); // tableau = {'a', 'k', 'o', 'q', 'r', 's'}

**int res = Arrays.binarySearch(tableau, 'k');**

Les objets peuvent être recherchés en fonction de leur ordre naturel ou par rapport à un comparateur.

Les méthodes *fill()* affectent un valeur donnée à tous ou à un intervalle des éléments d'un tableau. Chaque type d'élément possède un jeu de méthodes à l'instar des méthodes précitées.

double tableau = new double[10];

**Arrays.fill(tableau, 0.0);**

String[] tableau = new String[100];

**Arrays.fill(tableau, "");**

* Les méthodes *equals()* ont la fonction de tester l'égalité entre deux tableaux de même type. Si l'égalité des tableaux est effectives, la méthode *equals()* retourne *true*, sinon elle renvoie *false*.
* Une égalité effective est tributaire d'un nombre d'éléments équivalent et d'une correspondance stricte entre les paires d'éléments (ta[1] = tb[1], ta[2] = tb[2], etc..). Ainsi, pour assurer un même ordre aux deux tableaux à comparer, il peut être opportun de les trier préalablement.

float[] tab1 = {0.2f, 0.5f, 0.9f, 1.54f, 7.2f, 4.5f };

float[] tab2 = {0.2f, 0.5f, 0.9f, 1.54f, 7.2f, 4.5f};

float[] tab3 = {1.54f, 7.2f, 0.2f, 0.9f, 0.5f, 4.5f};

**boolean res = Arrays.equals(tab1, tab2);** // retourne true

**boolean res = Arrays.equals(tab1, tab3);** // retourne false; comparaison des éléments un à un

Arrays.sort(tab3);

**boolean res = Arrays.equals(tab1, tab3);** // retourne true

La méthode *asList()* constitue un pont entre les tableaux et les collections Java. En effet, cette méthode retourne une liste dont les éléments seront ceux du tableau spécifié en argument.

**List L = Arrays.asList(tableau);**

**Exemple treeSet**

|  |  |
| --- | --- |
| **import** java.lang.\*;  **public** **class** point **implements** Comparable{  **private** **int** x ;  **private** **int** y ;  **public** point( **int** x , **int** y ){  **this**.x = x ;  **this**.y = y ;  }  **public** **void** affiche(){  System.*out*.println("x="+ x+"et y="+y);  }  **public** **int** getX(){  **return** x ;  }  **public** **int** getY(){  **return** y ;  }  **public** **int** compareTo(Object o){  point p = (point)o;  **if**(**this**.y<p.y)  **return** -1;  **else**  **if**(**this**.y>p.y)  **return** 1;  **else**  **return** 0;  }} | public static void main(String[] args) {  point p;  int i;  //Comparator c = new comparateur();  TreeSet s=new TreeSet();  point t[] =new point [3];  t[0]=new point(14,800);  t[1]=new point(1000,762);  t[2]=new point(121,1000);  for(i=0;i<t.length;i++)  s.add(t[i]);    Iterator it = s.iterator();  while(it.hasNext())  {  p=(point)it.next();  p.affiche();  }  }  } |