|  |
| --- |
| **API Collections en JAVA** |
| **Présenté par :** Djamel **MOUCHENE** |



Propriétaire

High Tech Compass

**Versions**

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Auteur | Modifications |
| V 1.0 | 23/10/2017 | Djamel MOUCHENE | Version Initiale |
|  |  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Table ds matières

[**API Collections en JAVA** 0](#_Toc519775814)

[**Présenté par :** Djamel **MOUCHENE** 0](#_Toc519775815)

[**Versions** 1](#_Toc519775816)

[Principe 1](#_Toc519775817)

[Utilisation 1](#_Toc519775818)

[Listes (List) 1](#_Toc519775819)

[Méthodes sur les listes 1](#_Toc519775820)

[Listes chaînées (LinkedList) 2](#_Toc519775821)

[Tableau redimensionnable (ArrayList) 2](#_Toc519775822)

[Tableau redimensionnable (Vector) 2](#_Toc519775823)

[Files (Queue) 3](#_Toc519775824)

[Piles (Stack) 3](#_Toc519775825)

[Ensembles (Set) 4](#_Toc519775826)

[Les différentes implémentations 5](#_Toc519775827)

# Principe

Les collections en Java sont des classes permettant de manipuler les structures de données usuelles : listes, piles, files (ou queues).

Java collection

Une manière standard de représenter une structure de données en Java consiste à regrouper dans une interface l'ensemble des noms des opérations applicables sur celle-ci (ajout, suppression, effacement, etc.), c'est-à-dire l'ensemble des opérations mentionnées dans le type de données abstrait implémenté par cette structure (sa spécification, indépendamment du choix d'implémentation). Cette interface sera implémentée par chaque classe représentant cette sorte de structure : par exemple, l'interface List regroupe un ensemble de noms de méthodes génériques permettant la manipulation de listes, et est implémentée à la fois par les classes concrètes ArrayList (implémentation des listes par tableaux extensibles) et LinkedList (implémentation des listes par chaînage).

Les interfaces et classes citées dans les exemples ci-dessous sont livrées dans le SDK standard. Elles se trouvent dans le package java.util.

# Utilisation

List<String> ma\_liste = new LinkedList<String>();

Dans cet exemple, List est une interface qui implémente les listes (collection d'éléments ordonnés, éventuellement répétés), on a choisi d'utiliser une liste chaînée pour représenter la liste en mémoire. LinkedList est la classe dont les éléments de la collection seront des instances.

Le <String> entre chevrons représente le type générique.

# Listes (List)

Cette interface est implémentée par un certain nombre de collections, et garantit que ces classes implémenteront l'ensemble des méthodes. Elle dérive de l'interface Collection. Les éléments sont indexés (i.e. numérotés de la même façon qu'un tableau est indicé).

## Méthodes sur les listes

Type Méthode[1] Rôle

boolean add(int index, Object o) Ajouter un objet à l'index indiqué.

boolean addAll(int index, Collection c) Ajouter tous les objets d'une autre collection à l'index indiqué.

Object get(int index) Retourner l'objet à l'index indiqué.

int indexOf(Object o) Retourner le premier index de l'objet indiqué.

int lastIndexOf(Object o) Retourner le dernier index de l'objet indiqué.

Object remove(int index) Supprimer l'objet à l'index indiqué.

Object set(int index, Object o) Remplacer l'objet à l'index indiqué. L'objet précédent est retourné.

int size() Retourner le nombre d'éléments de la liste.

List subList(int fromIndex,int toIndex) Retourner une sous-liste de celle-ci.

Les différentes implémentations

On peut utiliser des tableaux redimensionnables (ArrayList) ou des listes chaînées (LinkedList)

## Listes chaînées (LinkedList)

Cette classe implémente l'interface List en chaînant les éléments (liste doublement chaînée).

Les méthodes ajoutées sont :

***void addFirst(Object o)*** : Ajoute un élément en début de liste.

***void addLast(Object o)*** : Ajoute un élément en fin de liste.

***Object getFirst()***

Retourne l'élément en début de liste.

***Object getLast()***

Retourne l'élément en fin de liste.

***Object removeFirst()***

Supprime et retourne l'élément en début de liste.

***Object removeLast()***

Supprime et retourne l'élément en fin de liste.

## Tableau redimensionnable (ArrayList)

Cette classe est un tableau dont la taille croit lorsque des éléments sont ajoutés.

## Tableau redimensionnable (Vector)

Cette classe est un tableau dont la taille croit lorsque des éléments sont ajoutés. Cette classe implémente les méthodes de l'interface List et les suivantes :

int indexOf(Object o,int index)

Retourne l'index de l'objet indiqué, en partant de l'index indiqué.

int lastIndexOf(Object o,int index)

Retourne l'index de l'objet indiqué, en partant de l'index indiqué et en allant vers le début (index 0).

void setSize(int newSize)

Tronquer/Agrandir le vecteur à la taille indiquée.

Cette classe a été créée avant la classe ArrayList. Elle est synchronisée : durant l'appel à une méthode de cette classe par un thread, un autre thread ne peut modifier le tableau.

# Files (Queue)

Files avec priorités (PriorityQueue)

Pour utiliser une file avec priorités, les éléments doivent être des instances d'une classe qui implémente l'interface Comparator. Il faudra écrire la méthode int compare(Object, Object) qui compare les deux instances et renvoie un entier.

Si vous ne voulez, ou ne pouvez pas, modifier la classe qui décrit les éléments, vous pouvez créer, dans la classe où vous utilisez votre file avec priorité, une classe interne qui hérite de la classe des éléments et implémente l'interface Comparator. Exemple, si vous voulez créer une file avec priorité de Bidules :



# Piles (Stack)

Une pile contient une liste d'objets. Il est possible d'ajouter un objet au sommet de la pile (empiler, ou push en anglais), et de retirer l'objet situé au sommet de la pile (dépiler, ou pop en anglais).

**Exemple**: En partant d'une pile vide, on effectue les opérations suivantes :

|  |  |
| --- | --- |
| **État de la pile** | **Opération** |
| (vide) | empiler A |
| A | empiler B |
| A B | empiler C |
| A B C | dépiler -> C |
| A B | dépiler -> B |
| A | empiler D |
| A D | dépiler -> A : ERREUR |
| AD | dépiler -> B |
| A | dépiler -> A |
| (vide) |  |

La classe Stack est une implémentation de pile qui dérive de la classe Vector et ajoute les méthodes suivantes pour gérer les objets comme une pile :

***boolean empty()***

Retourne vrai si la pile est vide.

***Object peek()***

Retourne l'objet au sommet de la pile sans l'enlever.

***Object pop()***

Retourne l'objet au sommet de la pile et l'enlève.

***Object push(Object o)***

Ajoute l'objet au sommet de la pile.

***int search(Object o)***

Retourne l'index de l'objet depuis le sommet de la pile (1 = sommet de la pile, -1 = non trouvé).

# Ensembles (Set)

Conformément à l'idée mathématique, les ensembles représentent plusieurs éléments non triés, sans répétitions. Les ajouts d'éléments déjà présents sont donc ignorés. Un élément est déjà présent si un test equals sur un des éléments de l'ensemble renvoie vrai.

Cette interface est implémentée par un certain nombre de collections, et garantit que ces classes implémenteront l'ensemble des méthodes. Elle dérive de l'interface Collection, sans ajouter de nouvelles méthodes. Elle sert seulement à indiquer informellement que la collection implémentant cette interface ne contient aucun doublon d'objet (objets comparés par la méthode equals). Cette interface correspond donc aux ensembles mathématiques.

## Les différentes implémentations

La classe HashSet implémente l'interface Set en utilisant une table de hachage.

TreeSet utilise un un arbre de recherche. Pour pouvoir utiliser un TreeSet, il faut que les éléments soit comparables. Cette fonction est plus lente que HashSet[2].

LinkedHashSet diffère de HashSet car il maintient une liste doublement liée à travers toutes ses entrées, permettant de retrouver l'ordre d'insertion (mais pas pour les réinsertions).

Ensembles triés (SortedSet)

Les ensembles triés sont identiques aux ensembles simples excepté qu'ils peuvent être triés par défaut à leur création selon un tri dit naturel ou selon un motif de tri. La méthode comparator() de cette interface permet de retourner le motif de tri utilisé ou retourne null si le tri est effectué de façon naturelle en fonction du type des données.

Range view

Permet des opérations sur les plages d'ensembles triés.

Endpoints

Renvoie le premier ou dernier élément d'un ensemble trié.

Comparator access

Renvoie le comparateur utilisé pour classer l'ensemble.

Tableaux associatifs (Map)

Cette interface est implémentée par les collections qui associent une clé à un objet. L'accès aux objets est donc effectué par une clé unique.

Il est possible d'utiliser n'importe quelle instance de classe comme clé. Cependant si cette classe ne possède pas les propriétés nécessaires, il faut utiliser exactement la même instance pour accéder à une valeur (voir Objets comparables et clés).

Les principales méthodes de cette interface sont :

***void clear()***

Vider la collection.

***boolean containsKey(Object key)***

Teste si la clé existe, c'est à dire associée à une valeur.

***boolean containsValue(Object value)***

Teste si la valeur existe.

***Set entrySet()***

Retourne l'ensemble des associations clés-valeurs.

***Set keySet()***

Retourne l'ensemble des clés.

***Collection values()***

Retourne la collection de valeurs.

***Object put(Object key, Object value)***

Associe la clé à la valeur spécifiée, et retourne la valeur précédemment associée.

***boolean putAll(Map m)***

Ajouter tous les objets d'une autre collection à celle-ci.

***Object get(Object key)***

Retourne la valeur associée à la clé spécifiée, ou null si non trouvé.

***Object remove(Object key)***

Supprime l'objet associé à la clé, et retourne cet objet.

***boolean isEmpty()***

Tester si la collection est vide.

***int size()***

Retourne le nombre de clés.

Les implémentations

La classe Hashtable implémente l'interface Map de manière synchronisée.

La classe HashMap implémente l'interface Map, et permet d'utiliser la clé null.

Les tableaux triés (SortedMap)

Exactement comme SortedSet :

Range view

Permet des opérations sur les plages de tableaux triés.

Endpoints

Renvoie le premier ou dernier élément d'un tableau trié.

Comparator access

Renvoie le comparateur utilisé pour classer le tableau.

Parcourir une collection

Les itérateurs

Les itérateurs sont des outils puissants pour parcourir le contenu d'une collection.

List<String> ma\_liste = new LinkedList<String>();

ma\_liste.add("Bonjour");

ma\_liste.add(" le ");

ma\_liste.add("monde");

Iterator<String> it = ma\_liste.iterator(); // On paramètre Iterator par le type des éléments de la collection qui sera parcourue

while (it.hasNext()) {

System.out.println(it.next()); // Affiche "Bonjour le monde"

}

Attention, l'appel de next() renvoie l'élément courant dans le parcours et passe l'itérateur à l'élément suivant.

Pour pouvoir itérer sur une collection, il faut qu'elle soit itérable, c'est à dire qu'elle hérite de Iterable. C'est le cas la plupart du temps mais il y a des exceptions (HashMap).

L'utilisation des itérateurs requiert la classe Iterator définie dans le package java.util.

Boucles « for each »

// Parcours d'une collection

Collection<ObjetX> collection = ......;

for(ObjetX objetX : collection){

objetX.methodeY(p1,p2);

}

Attention aux parcours des Map

Le code suivant convient tout a fait si vous avez besoin de parcourir les clés de la Map sans vous préoccuper des valeurs

Map<Key, Value> map;

for (Key key : map.keySet()) {

// ...

}

Le code suivant est à proscrire si vous parcourrez l'ensemble des clés et des valeurs qui leurs sont associées

for (Key key : map.keySet()) {

// l'appel de get engendre un temps coûteux en accès à chaque itération

Value value = map.get(key);

}

Dans le cas précédent, préférez le code suivant

for (Map.Entry<Key, Value> entry : map.entrySet()) {

Key key = entry.getKey();

Value value = entry.getValue();

// ...

}

Interface commune

La plupart des structures évoquées ci-dessus implémentent l'interface Collection et possèdent donc un ensemble de méthodes communes. Les principales méthodes sont :

***boolean add(Object o)***

Ajouter un objet à la collection.

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

Retirer un objet de la collection.

***boolean contains(Object o)***

Tester si la collection contient l'objet indiqué.

***boolean addAll(Collection c)***

Ajouter tous les objets d'une autre collection à celle-ci.

***boolean removeAll(Collection c)***

Retirer tous les objets d'une autre collection de celle-ci.

***boolean retainAll(Collection c)***

Retirer tous les objets qui ne sont pas dans la collection spécifiée de celle-ci. À la fin les deux collections contiennent les mêmes objets.

***boolean containsAll(Collection c)***

Tester si la collection contient tous les objets de la collection indiquée.

***void clear()***

Vider la collection.

***boolean isEmpty()***

Tester si la collection est vide.

***Iterator iterator()***

Retourne un itérateur permettant de faire une boucle sur tous les objets contenus dans la collection.

***int size()***

Retourne le nombre d'objets de la collection.

***Object[] toArray()***

Convertit la collection en tableau d'objets.

***Object[] toArray(Object[] a)***

Convertit la collection en tableau d'objets de classe spécifiée. Si le tableau fourni n'est pas assez grand, un nouveau tableau est alloué en utilisant la même classe d'élément.

Tableau récapitulatif

Interface Implémentations

Tableau dynamique Chaînage Arborescence[3] Table de hachage

Liste (List) ArrayList, Vector LinkedList

Ensemble (Set) TreeSet HashSet

File (Queue)

Tableau associatif (Map) HashMap

Synchronisation

Si une collection est accédée par plusieurs threads, il faut utiliser la synchronisation afin que les modifications soient cohérentes, c'est à dire exécutées sans être interrompues par une modification effectuée par un autre thread.

Une collection synchronisée garantit que l'appel d'une méthode de la collection sera effectuée sans qu'aucun autre thread ne modifie cette collection entre-temps. Pour obtenir une collection synchronisée, vous devez appeler une méthode de la classe Collections, dont le nom dépend du type de collection :

type variable = Collections.synchronizedtype( new type(...) );

Exemple :

Set myset = Collections.synchronizedSet( new HashSet() );

Cependant, le thread peut être interrompu entre deux appels de méthode. Dans ce cas, vous devez synchroniser l'ensemble des instructions appelant les méthodes de la collection (mot-clé synchronized). Voir le chapitre "Threads et synchronisation". Autrement dit, les listes ne sont pas thread-safe, aucune méthode n'est atomique. Les sources de ces classes sont d'ailleurs consultables si vous avez accès au JDK.

La solution précédente peut résoudre tous les problèmes de synchronisation, par exemple celui ci :

Map<String,String> map=new HashMap<String,String>();

map.put("toto", "valeur exemple 1");

map.put("titi", "valeur exemple 1");

...

synchronized {

if (!map.containsKey("titi"))

bd.put("titi", "valeur exemple 3");

}

Une modification ne peut avoir lieu, par un autre thread, entre le containsKey et le put. Mais cette solution est peu efficace en réalité car elle contraint à protéger tous les accès à la liste en lecture comme en écriture, alors que les accès en lectures multiples n'ont pas à être bloquants.

La solution la plus efficace est d'utiliser les listes de la bibliothèque java.util.concurrent qui possède un putIfAbsent. En outre, les méthodes de cette bibliothèque peuvent être considérées comme toujours plus rapides en mode multi-thread que celle d'une liste synchronisée,

Classes squelettes

Une interface possède souvent une longue série de méthodes à implémenter. Certaines sont implémentées généralement de la même manière par beaucoup de classes.

Plutôt que de répéter le code de ces méthodes dans les différentes implémentations, il est préférable de rassembler leur code dans une classe abstraite (car toutes les méthodes de l'interface ne sont pas implémentées).

Les classes de l'API Java utilisent de telles classes dont dérivent les différentes implémentations.

Ces classes sont :

AbstractCollection pour l'interface Collection,

AbstractList pour l'interface List,

AbstractSequentialList pour l'interface List,

AbstractSet pour l'interface Set,

AbstractMap pour l'interface Map.

Elles ne sont pas instanciables directement car abstraites, mais servent de classes de base. Il est utile de connaître ces classes pour rechercher dans la documentation, ou réaliser une implémentation alternative à celle proposée par l'API.