Simplon : Java

Nicolas Rousset

Vendredi 5 août

Structures classiques en Java

Ce sont principalement :

Les collections

- Queue
- List
- Set

Les maps

- Map

Les tableaux (analogues aux List)

Les collections

Elles ne peuvent pas stocker de type primitif, d'où des types spéciaux :

- Integer -> int
- Long -> long
- Float -> float

...

Les collections

Interface commune, elles ont notamment toutes les méthodes :

- add / addAll
- remove
- contains / retainsAll / removeAll
- itérations

Attention encore une fois à ce qu'elles ne sont pas aussi efficace suivant les structures !

Queue

C'est une file d'attente, comme lorsque vous allez à la poste !
Le premier arrivé est le premier servi, les queues servent essentiellement à traiter des taches (comme des messages webservices).

```
Queue<int> queue = new LinkedList<int>();

queue.add(1);
queue.add(2);
queue.add(3);

queue.poll(); // => renvoit 1
queue.poll(); // => renvoit 2
```

Elles sont très souvent ThreadSafe.
Elles existent car les ArrayList sont très peu efficace pour supprimer le premier élément (et les Set ne conservent pas l'ordre d'insertion)

List

C'est la collection la plus standard !

Elles conservent l'ordre d'insertion, tolèrent les doublons, et sont très efficaces pour ajouter des éléments et retrouver un élément à partir de leur position.

Par contre elles n'ont pas de moyen de retrouver des éléments à partir de leur valeur, elles sont très inefficaces pour toutes les opérations consistant à savoir si un élément est présent :

- retainAll
- contains
- containAll
- removeAll

Utilisez des ArrayList par défaut.

(Les array / tableau de type int[] ont les mêmes caractéristiques, la seule différence étant que leur taille est fixe)

Les Set

Au niveau de l'interface, c'est la même que la List, sauf que vous n'avez pas la méthode get.

Les éléments ne sont plus organisés selon leur ordre d'insertion, mais selon leur valeur. Vous ne pouvez pas avoir de doublons.

D'où les effets suivants :

- il est beaucoup plus rapide de vérifier la présence, l'absence d'un élément
- il est plus long d'insérer un élément ou de supprimer le dernier

Les Set (2)

L'insertion est plus lente, car le code ne peut pas juste le mettre à la fin, il doit le mettre "au bon endroit" pour pouvoir le retrouver facilement après.

Attention à la notion de doublons ! Elle dépend de l'implémentation de la classe (méthode equals / compareTo / hashCode())

Les Set : implémentations

Il en existe deux types principaux, HashSet et TreeSet.

- TreeSet => nécessite que l'objet soit Comparable (mais ne vous le dit pas à la compilation !!!)
- HashSet => pas de contrainte, souvent plus rapide

Le code suivant provoque une erreur lorsque vous essayez d'insérer un élément (que c'est moche ...)

```
Set<ItemAllType> set = new TreeSet<ItemAllType>();
```

Par contre, lorsque vous itérez sur un TreeSet, ils seront dans l'ordre, alors qu'avez un HashSet non.

Les Set : implémentations

Par défaut utilisez un HashSet.
Si vous avez des objets Comparable et que vous voulez itérez dessus dans l'ordre, utilisez un TreeSet.

Set VS ArrayList ??

En un mot, par défaut utilisez des ArrayList, si vous devez faire des contains / retainAll, utilisez des Set.

Exemples :

- filtre sur les marques, vous ne devez garder que les items de certains marques
- comptage du nombre de marque différentes
- comparer les marques communes ou non entre deux fichiers

Savoir si deux sets contiennent les mêmes éléments :

```
set1.size() == set2.size() && set1.containsAll( set2 );
```

Comparaison set / array

	Set	ArrayList /
Accès par index	Non	O(1)
Accès par valeur (contains)	O(log(n))	O(n)
Ajout d'élément	O(log(n))	O(1)
Ajout au début	Non	O(n)
Remove à la fin	Non	O(1)
Remove	O(log(n))	O(n)
Parcours	Ordre des éléments	ordre d'insertion
View sur sous partie	Non	Oui

Map

Pour l'instant, on a vu :

- List => accéder aux éléments par leur position
- Set => accéder aux éléments par leur valeur (pas vraiment un accès)

Comment faire si je veux trouver un élément à partir de son id ??

Map

Association clé / valeur

```
Map<Long, ItemAllType> map = new TreeMap< Long, ItemAllType >();

map.put( item.getId(), item );
map.get( item.getId() ); // retrouve rapidement mon Item
```

Même principe que le Set, sauf que les éléments sont organisés par rapport à une clé.

Pas de méthode add, il faut forcément ajouter une clé et une valeur.

Map (2)

A quoi ça sert ?
Accès par Id, par Titre, par ce qu'on veut.
Compter les éléments.

Chaque clé est unique.
En cas de put, si la clé est déjà présente, elle est écrasée.

Map (3)

Les méthodes sont déclinées sur les clés et sur les valeurs.

```
containsKey( ... )   ; // => très efficace
containsValue( ... ); // => très lent

for(Map.Entry<String, Integer> e: brandCounter.entrySet() )
{
    System.out.println( e.getKey() );
    System.out.println( e.getValue() );
}
```