JAVA : COLLECTIONS :

Working with Arrays and Collections in Java (Java SE 11 Developer Certification 1Z0-819)

I. Fundamentals :

• *What is a collection :*

Groupe d'élément qui ont une relation les uns avec les autres.

Ils sont indexés par numéro, position ou clef.

Les 4 premiers types sont : Listes, Set, Queues et Map.

• *Lists, Sets, Queues and Maps :*

Toutes = classe collections **SAUF** map.

Listes : La position est basée sur le numéro qui commence à 0, permet les duplicatas d'item, les performances diminuent en fonction de la taille.

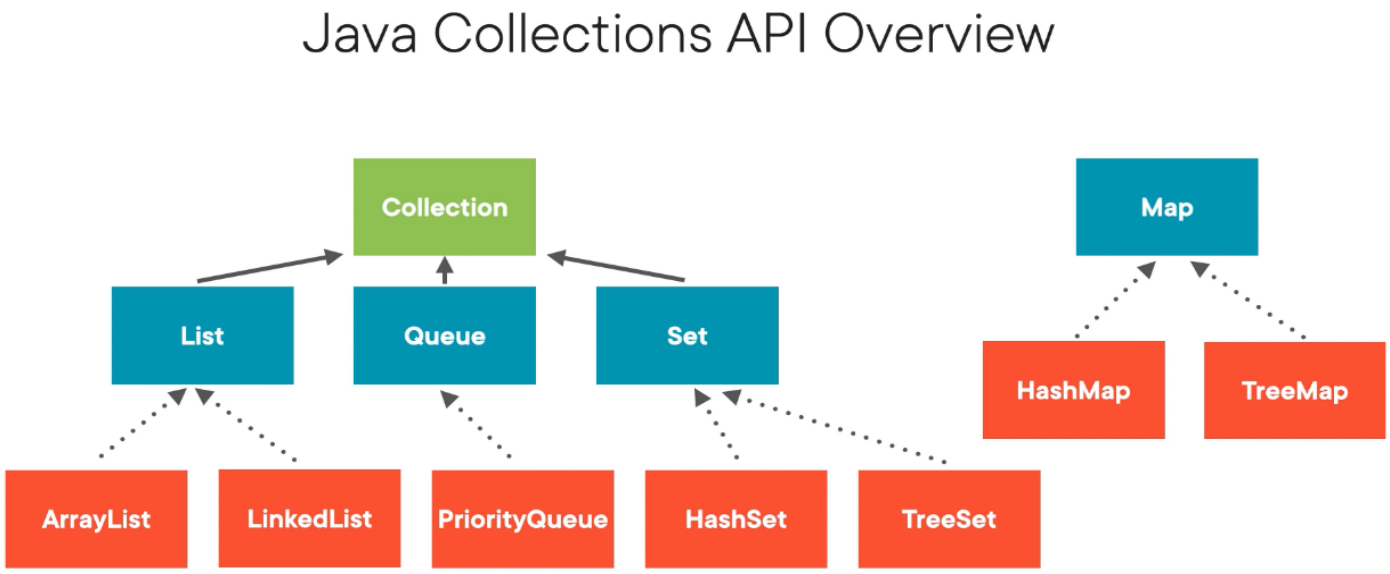
Note : Si l'index commence à zéro, c'est parceque les éléments sur placé les uns à coté des autres dans la mémoire. Si on a un int qui prend 4 bits, l'emplacement 2 sera calculé comme étant l'emplacement 1 plus 1 fois 4.

Sets : Quasi les mêmes que Listes (position basée sur un chiffre) mais tous les éléments doivent être **uniques**, donc pas de duplicata. Aussi les performances qui diminuent fonction de la taille.

Queue : Position basée sur l'ordre d'entrée (fifo), autorise les duplicatas et pareille performance, qui diminuent.

Map : Très différente, les éléments son basés sur une **CLEF.** Les clefs sont **UNIQUES**. Les performances ne diminuent pas avec la taille, car les items sont trouvés par clefs plutôt que par itération.

- ***JAVA Collection Overview :***

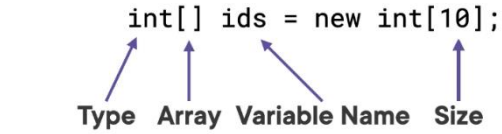


Java Type	Parent Interface Type	Allows null?	Elements Sorted?	Uses hashCode?	Uses compareTo?
ArrayList	List	Yes	No	No	No
LinkedList	List and Queue	Yes	No	No	No
HashSet	Set	Yes	No	Yes	No
TreeSet	Set	No	Yes	No	Yes
HashMap	Map	Yes	No	Yes	No
TreeMap	Map	No	Yes	No	Yes

II. Working with ARRAYS :

- ***Primitive Arrays :***

Déclaration du tableau le plus basique :




```
int[] ids = new int[] {54, 22, 37, 1, 99};
```

Ils doivent tous être déclaré **avec une taille** OU **avec des éléments**.

Tous les tableaux commencent leur index avec zéro.

On peut quand même déclarer le tableau comme ceci : donc juste les crochets et leur contenu peuvent servir à instancier. Cette forme est connue comme un tableau **ANONYME**.

```
int[] ids = {54,22,37,1,99}
```

- ***Arrays in Action :***

Pour afficher le contenu d'un array :

```
int []ids = new int[10];
System.out.println(ids[0]);
```

Cela va afficher **ZERO** car le tableau n'a aucun contenu et un primitif ne peut pas être un **NULL** donc Java attribue un zero par défaut.

Si je fais pareil mais avec STRING j'aurai un NULL à la place du zéro.

Pour déclarer, instancier et attribuer en même temps :

```
String[] instruments = new String[]{"guitar", "drums",
"bass"};
System.out.println(instruments[0]);
```

L'ordre de la déclaration n'a pas tellement d'importance, on peut mettre le nom, **AVANT** ou **APRES** les crochets, et on peut même en mettre plusieurs (ligne 3).

```
int []ids3;
int ids4[];
int[] ids6, ids7, ids8;
```

- ***Array Value Allocation :***

Si on a un primitif : c'est la valeur qui est stockée.

Si on a un objet : c'est la référence qui est stockée.

- ***Accessing and Iterating :***

Connaitre la longueur : nomArray.length

Itérer, et attention à faire un en moins que la taille car on commence à zéro : boucle for (int i=0 ; i<nomArray.length ;i++)

La boucle **forEach** est aussi valable pour les tableaux.

- ***Sorting Arrays :***

Les tableaux ont des méthodes plutôt primitives donc on peut faire appel à une autre classe et ses méthodes static pour s'en sortir : **ARRAYS**. Et la méthode **SORT**. (Cette méthode ne renvoie rien). Ici on trie simplement des objets (instrument est un tableau).

```
Arrays.sort(instruments);
```

On peut aussi imprimer le contenu facilement grâce à la méthode **toString** :

```
System.out.println(Arrays.toString(instruments));
```

- ***Searching Arrays :***

On peut utiliser la méthode **binarySearch** qui a comme paramètre : Un **tableau** composé soit de primitif, soit d'un objet et une **valeur**.Il y'a une autre version qui permet de délimiter les index dans lesquels on cherche.

```
int[] fibArray = new int[] {0, 1, 5, 2, 3, 1, 8, 13};
Arrays.sort(fibArray);
System.out.println(Arrays.binarySearch(fibArray, 3));
```

Dans ce cas j'obtiens -7 car les éléments **DOIVENT** être trié avant de faire une binarySearch.

Le résultat sera soit :

- L'index ou il se trouve.
- Un chiffre négatif si l'élément n'est pas dans un tableau trié.
- Un résultat imprévisible si le tableau n'est pas trié.

- ***Understanding Arrays Comparison :***

Il y'a 4 manières de comparer des tableaux :

- Array1.**equals**(Array2) : compare l'adresse. Pas spécialement utile avec des tableaux.
- Arrays.**equals**(Array1, Array2) : renvoi true s'il y'a les mêmes éléments et dans le même ordre.
- Arrays.**compare**(array1, array2) : compare s'ils sont plus grands, plus petits ou égaux. Renvoi un neg(plus petit), un zero(égaux) ou un positif(plus grand).
- Arrays.**mismatch**(array1, array2) : Comparaison pour savoir ou les tableaux commencent à différer. Si égaux : -1 si pas : index de la première différence.

Les deux tableaux doivent être de même type.

- ***VarArgs and the Java main method :***

Le tableau déclaré dans la méthode main peut être configuré dans intelliJ via l'icône de clef anglaise qui se trouve en bas à gauche lorsqu'on lance un run du programme. On peut alors écrire des choses dans la ligne programme arguments et si on affiche args on verra ce qu'on a écrit.

- ***Tableaux multidimensionnels :***

Déclaration et initialisation : (on peut mettre autant de crochets que l'on souhaite)

```
int[][] multiArray = new int[3][3];
```

On peut aussi faire comme ceci :

```
String[][] multiArray2 = new String[3][10];
```

Ce qui veut dire que pour chaque index, on a 10 indexes.

Exemple d'initialisation avec des éléments :

```
String[][] westCoastCitiesGrouped = new String[][]{
        {"LA", "San Francisco", "San Diego"},
        {"Seattle", "Tacoma"},
        {"Portland"},
};
```

Pour itérer dedans il faut autant de boucles qu'on a de tableaux :

```
for(int i = 0; i < westCoastCitiesGrouped.length; i++) {
    for(int ii = 0; ii < westCoastCitiesGrouped[i].length;
ii++) {
```

A noter qu'on peut aussi faire deux forEach dans notre cas.

III. Working with Lists :

- ***Intro the ArrayList :***

Propriété principale : elles sont **dynamiques** ! Donc pas de limite du nombre d'éléments et elles ne peuvent pas prendre de type PRIMITIFS.

Ici, **6 différentes** formes de déclaration et d'initialisation, la deuxième avec le nombre de place, la troisième, apd d'une liste existante. Ces déclarations sont pré-JAVA 5, donc **sans générique**.

```
ArrayList arrayList = new ArrayList();
ArrayList arrayList2 = new ArrayList(100);
ArrayList arrayList3 = new ArrayList(arrayList2);
```

Version plus moderne :

Note, il vaut mieux déclarer une liste par sa classe parente (2eme et 3eme déclaration). Les diamants sont optionnels.

```
ArrayList<String> colors = new ArrayList<String>();
List arrayList4 = new ArrayList();
Collection arrayList5 = new ArrayList();
```

- ***Using the ArrayList :***

- Ajouter un élément, 2 manières possibles, la deuxième utilisant un int pour indiquer l'index d'emplacement. .

```
colors.add("blue");
colors.add(1, "orange");
```

L'itération se fera sur grâce à la méthode size (length étant propre aux tableaux) :

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

Possibilité aussi d'utiliser **forEach**.

Depuis JAVA 8 nouvelle forme possible utilisant les lambdas :

```
colors.forEach(color -> {
    System.out.println(color);
});
```

- Supprimer un élément, deux possibilités : par le contenu, par le numéro d'index :

```
colors.remove("orange");
colors.remove(0);
```

Attention à ne pas supprimer des éléments avec une loop, sinon on va vers une exception.

- ***Array to ArrayList (conversion) :***

A. Convertir une liste en tableau basique :

Normalement on a des Strings dans la liste (colors) mais quand on converti, on utilise le type **Object** (et non String) sinon on a une erreur de compilation. (ligne 1).

Si je veux le convertir en String je dois le faire selon la deuxième méthode.

```
Object[] colorObjArray = colors.toArray();
String[] colorArray = colors.toArray(new String[0]);
```

B. Convertir un tableau en liste :

La conversion dans ce sens est beaucoup plus simple. La première ligne ici est simplement l'initialisation du tableau.

```
String[] shapes = new String[]{"square", "circle",
"triangle"};
List shapeList = Arrays.asList(shapes);
```

Si on ne sait pas quel type de liste c'est : nomListe.getClass().getName().

Attention que quand on converti on obtient pas une vraie ArrayList mais une conversion ce qui donne deux résultats différents pour chacun si on veut voir le nom de la classe :

La conversion : java.util.Arrays$ArrayList

L'original : java.util.ArrayList

Le fait est, que le tableau converti ne permet pas tout à fait de faire les même chose qu'une liste originale. C'est du au fait que quand on converti il renvoi une liste à la taille **FIXE** qui match avec notre tableau. On peut alors modifier les éléments mais on ne peut ni les supprimer ni en ajouter. Remove ou Add, lancera alors dans ce cas une exception.

On peut aussi utiliser une autre méthode de conversion qui utilise une méthode statique de list :

```
List shapeList2 = List.of(shapes);
```

Cette fois-ci le résultat est tout à fait **immutable**. Donc aucune modification possible.

La meilleure manière de faire donc est de le faire **à l'ancienne**, via une **boucle** :

```
List<String> shapeList3 = new ArrayList<>();
for(String shape: shapes) {
    shapeList3.add(shape);
}
```

- ***Sorting List :***

La méthode de tri utilisée pour les listes, provient de la classe d'aide **COLLECTIONS** à ne pas confondre avec l'interface Collection (au singulier)

On utilise la méthode **sort()** sur une liste de nombres.

```
Collections.sort(numberList);
```

Le résultat pour des nbr croissants sera un tri du plus petit au plus grand.

- ***Comparable and comparator :***

Java propose deux moyens différents de configurer un algorithme de tri :

Comparable : c'est une interface qui comprend une méthode compareTo(T o) qui comprend un argument, et un générique <T>. La comparaison à cette interface est censée être implémentée **par l'objet** auquel on souhaite ajouter une capacité de tri.

Comparator : c'est une interface FONCTIONELLE. Il est destinée à être utilisé pour trier une classe sur laquelle on a pas de contrôle mais qu'on veut trier, ou pour fonction lambda. La méthode comprend deux paramètres.

```
public interface Comparable<T> {
    public int compareTo(T o);
}


@FunctionalInterface
public interface Comparator<T> {
   int compare(T o1, T o2);
}
```

Fonctionnement :

A. Le **comparable** : quand je créé une classe j'implémente l'interface. Ici par exemple on créé une classe mountain :

```
public class Mountain implements Comparable<Mountain> {
    }
```

Il faudra alors implémenter la méthode et la configurer.
Si les objets sont égaux il renverra un zéro, si inférieur un -1, si supérieur un 1. Je précise aussi les attributs que je dois comparer, ici la hauteur:

```
public int compareTo(Mountain o) {
    return this.height - o.height;
}
```

Lorsqu'on stockera toutes ces montagnes dans une liste on pourra alors faire un tri avec la méthode sort() :
Collections.*sort*(mountains);
Si je n'implémente pas la méthode dans la classe il y'aura une erreur de compilation : *no suitable method found for sort*

B. Le **comparator** : Ici on implémente la méthode de comparaison en tant que classe **interne anonyme.**

```
Comparator<Mountain> mountainComparator = new
Comparator<Mountain>() {
    @Override
    public int compare(Mountain o1, Mountain o2) {
        return o2.getHeight() - o1.getHeight();
    }
};
```

Le comparator prend deux objets, le premier et le deuxième. Ensuite on configure la valeur de retour.

On peut alors utiliser la méthode sort(), mais cette fois-ci elle prend **DEUX** arguments, la liste mountains, et l'objet comparator.

```
Collections.sort(mountains, mountainComparator);
```

On peut aussi écrire le comparator avec une fonction **lambda** ce qui économise de l'espace :

```
Comparator<Mountain> mountainComparator2 = (m1, m2) ->
        m2.getHeight() - m1.getHeight();
```

Comme le comparable il est possible d'implémenter le comparator dans une clase :

```
public class AgeComparator implements Comparator<Person>
{
    public int compare(final Person left, final Person right)
    {
        return Integer.compare(left.getAge(), right.getAge());
    }
}
```

IV. Working with SETS and MAPS :

- ***Introduction :***

Différence entre un Set et une List : Il ne peut pas contenir d'éléments en double et l'ensemble ne comprend pas d'ordre implicite (donc pas d'index).

- ***Using HashSet and TreeSet :***

Création de deux Sets :

```
Set<Integer> primeNumbers1 = new HashSet<>();
Set<Integer> primeNumbers2 = new TreeSet<>();
```

Si on ajoute des nombres dans cet ordre : 71,61,41,1. Dans un set de type HASH, quand on fera un print des éléments de la liste, ils n'apparaitront pas dans le même ordre qu'on les a ajoutés. Ils ont été stockés selon un ordre **ALEATOIRE**.

Si on fait la même chose avec le set de type TREE, on aura un ordre TRIE. Le TreeSet trie les éléments au fur et à mesure qu'ils sont ajoutés.

Même si je change l'ordre dans lequel j'ajoute ces nombres, j'aurai ces deux résultats :

```
HashSet :1, 71, 61,41
TreeSet : 1, 41, 61, 71
```

Si je veux ajouter un élément qui existe déjà, il renvoi un false implicite mais qui n'entraine pas d'exception :

La première ligne affichera true, la deuxième false.

```
System.out.println(primeNumbers1.add(5));
System.out.println(primeNumbers1.add(5));
```

Globalement les autres méthodes sont similaires à celle de List.

- ***The HashSet hashCode method :***

Quand on ajoute un élément dans un HashSet, en vérité on le fait en tant que pair clé/valeur, la valeur étant l'élément, et la clef est un **HASHAGE**.

Ici on voit ce que donne un HashSet de String de nom de couleurs :

Colors	"Red"	"Yellow"	"Blue"
Hash Code	82033	-1650372460	2073722

La classe de String a un code de hashage implémenté de sorte que chaque objet String créé un nombre int **unique**.

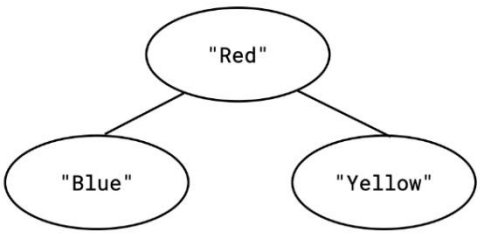
Pour vérifier qu'un objet existe déjà ou pas quand on l'ajoute, le hashSet passe par le hashCode et vérifie que ce nombre n'existe pas encore.

Si l'on créé un objet personnalisé, il est possible de créer une implémentation de hashCode valide pour notre objet. (Pas plus de précision dans la vidéo donc à creuser, voir autre vidéo collection pluralSight).

- ***The treeSet compareTo method :***

Pour trier, ce type de Set n'utilise pas le hashCode mais utilise les méthodes déjà vues avant avec les listes.

Si on ajoute les couleurs dans un TreeSet on peut représenter cela comme ça :



On obtient une forme d'arbre, d'où le nom. Si les caractère Alpha de l'éléments ajoutés sont inférieurs à celle de « red » ils vont dans la branche gauche.

S'ils sont supérieurs, ils vont dans celle de droite.

La méthode utilisée est un comparable : public int compareTo(T o). Pas besoin avec ce Set d'utiliser la méthode sort() car cela se fait au fur et à mesure qu'on ajoute des éléments.

Du coup est-il possible de trier un HashSet ? **NON**, la méthode sort() de la classe Collection ne fonctionnera pas. Il faut pour cela utiliser un TreeSet.

- ***The Map interface :***

1. The HashMap :

Très similaire au HashSet, car les deux contiennent des pairs valeur/clé. Sauf qu'ici il faut les définir nous-même. Autre différence, si les clés doivent être uniques, les valeurs peuvent être dupliquées.

Il y'a 4 manières de les déclarer, avec ou sans génériques, dans la classe parente ou dans la classe effective (Map ou HashMap). Dans les génériques, on définit d'abord le type de la clé ensuite le type de la valeur.

```
HashMap countries = new HashMap();
HashMap<Integer, String> countries2 = new HashMap<>();
Map languages = new HashMap();
Map<String, String> languages2 = new HashMap<>();
```

Ajouter un élément, on utilise la méthode **put()**, suivi de la clé et de la valeur en argument :

```
countries2.put(840, "USA");
countries2.put(484, "MEX");
countries2.put(124, "CAN");
```

Récupérer un élément, grâce à la méthode get() suivi de la clé en argument :

```
System.out.println(countries2.get(840));
```

Si on veut **enlever un élément**, idem, toujours avec la clef.

```
countries2.remove(840);
```

Si l'on veut **récupèrer** toutes les **clefs** de la map on utilise keySet qui renvoie un Set.

```
countries2.keySet();
```

Si l'on ajoute deux valeurs différentes avec une clé identique, cela effacera la première valeur. Il vaut donc toujours mieux **VERIFIER** si la clef existe déjà lorsqu'on ajoute une valeur.

```
countries2.put(124, "CAN");
countries2.put(124, "ABC");
```

Pour vérifier si la clef ou la valeur, existent déjà ou pas, il y'a deux méthodes :

```
countries2.containsKey(840);
countries2.containsValue("USA");
```

La méthode values() renvoi toutes les valeurs sous forme d'une Collection qui peut être ensuite utilisé comme liste.

```
countries2.values();
```

Les HashMap **peuvent** contenir la valeur **NULL**, tout comme les valeurs. Ici les deux écritures sont valides. Attention que le principe d'unicité s'applique aussi à une clef null, il ne peut y en avoir qu'une.

```
countries2.put(-1, null);
countries2.put(null, null);
```

2. The treeMap :

La déclaration suit le même principe que la HashMap.

```
TreeMap<Integer, String> planets = new TreeMap<>();
Map<String, String> englishSpanish = new TreeMap<>();
```

La différence principale vient du fait qu'encore une fois les éléments sont triés dès leur ajout comme dans un SetTree selon leur clef.

```
planets.put(3, "Earth");
planets.put(2, "Venus");
planets.put(4, "Mars");
```

Si on affiche le contenu :

```
System.out.println(planets.keySet().toString());
```

On obtiendra 2,3,4.

On peut tout à fait ajouter une valeur null car celles-ci ne sont pas vraiment traités étant donné que le tri se fait sur la clef.

```
planets.put(20, null);
```

Par contre ce code entrainera une exception car la TreeMap ne peut pas trier sur un null.

```
planets.put(null, null);
```