JAVA 04 : LES COLLECTIONS

LES TABLEAUX

Les tableaux sont des structures de données regroupant plusieurs valeurs de même type.
Les tableaux constituent des collections d'informations homogènes, c'est à dire, de valeurs primitives ou d'objets de même type. Les éléments d'un tableau peuvent être :

- des primitives (float, int, char, etc.)
- des références d'objets (String, Object)
- des références de tableaux.

La taille d'un tableau est fixée d'une façon permanente suite à la déclaration du tableau et à l'allocation de ressources systèmes pour ce dernier.
La déclaration des tableaux s'effectuent par l'intermédiaire de crochets ([...]).

```
type identificateur[];
type[] identificateur;
double resultats[];
char[] lettres;
String[] employes;
```

Il est possible de placer les crochets après l'identificateur ou après le type. La taille d'un tableau n'est spécifié qu'à partir du moment de sa création.
La création s'accomplit en utilisant l'opérateur **new.**

```
identificateur = new type[taille];
notes = new int[10];
```

la variable notes fait référence à un tableau de 10 valeurs entières. Par défaut, les valeurs de chaque élément d'un tableau sont initialisés à :

0 pour des entiers (int, short, ...),
0.0 pour des nombres à virgule flottante (double, float, ...),
u0000 pour des caractères (char),
false pour des booléens (boolean),
null pour des objets (Object, String).

Les tableaux peuvent être créés et initialisés par l'intermédiaire d'une liste de valeurs séparées par une virgule et compris entre des accolades.

```
type identificateur[] = {valeur, ..., valeurN};
int notes[] = {10, 9, 12, 14, 16, 15, 17, 20, 18};
```

L'accès aux éléments d'un tableau est réalisé en utilisant des valeurs d'indices qui sont les positions des éléments dans le tableau, en sachant que le premier commence à l'indice 0.

```
valeur = identificateur[indice];
premiereNote = notes[0]; // = 10
sixiemeNote = notes[5]; // = 15
```

La redéfinition d'un tableau entraîne la perte de l'ancienne structure avec toutes ses valeurs.

```
notes = int[50];
// Création d'un nouveau tableau de 50 valeurs //entières
initialisées par défaut à 0
```

La boucle for permet de parcourir un tableau d'une manière automatique de la même façon qu'en PHP.
D'autre part, la propriété **length** associée à un objet de type Array, fournit la taille d'un tableau.

```
for(int i = 0; i < notes.length; i++) {
            System.out.println(notes[i] );
          }
```

La syntaxe de l'instruction for peut être simplifiée pour énumérer un à un les éléments d'une collection (tableau, ArrayList...) :

```
for(int note : notes) {
        System.out.println(notes;
     }
```

La taille d'un tableau est fixée lors de sa construction et ne peut plus être modifiée. La seule solution consiste à créer un nouveau tableau plus grand et à copier les anciennes valeurs dans le nouveau tableau. Pour effectuer la copie de tableau, on utilise la méthode arraycopy.

```
/** la méthode simple */
int[] nouveau = new int[longueur];
System.arraycopy(vieuxTableau,0,nouveau,0,vieuxTableau.length);
/** Note : On place ici les anciennes valeurs au début du nouveau tableau. */
```

Remarque : on utilisera les tableaux en JAVA lorsque l'on connait exactement le nombre d'éléments à stocker , par exemple les mois de l'année, les jours de la semaine, toutes les lettres de l'alphabet. Si on ne connait pas à l'avance la taille du tableau, ou lorsque les ajouts, insertions, modifications vont être importantes, il est préférable d'utiliser des « outils » plus performants.

Vector - ArrayList

Java fournit la classe Vector et ArrayList dans son package java.util. Un objet de la classe Vector ou ArrayList est similaire à un tableau puisqu'il permet de stocker plusieurs valeurs.
Une différence importante entre les tableaux est que ces derniers gèrent implicitement leur taille, celle-ci peut changer automatiquement lors de l'exécution du programme s'il manque de place pour un nouvel élément. Par contre, l'implémentation de ces classes utilise (implicitement) un tableau. Lorsque nous insérons un nouvel élément ailleurs qu'à la fin, tous ceux qui le suivront seront d'abord décalés, un à un, d'une position dans le tableau, ce qui ralenti l'exécution du programme.

La déclaration d'un Vector ou d'un ArrayList s'effectue de la manière suivante :

```
Vector <type de variable> nom_vecteur = new Vector<type de variable>();
List <type variable> nom_liste = new ArrayList<type variable>();
```

Exemple : on souhaite stocker le nom des pays dans un ArrayList et la population dans un Vector

```
//création d'un ArrayList nommé pays pouvant stocker des String
List <String> pays = new ArrayList<String>();

//création d'un Vector population pouvant stocker des entiers
Vector <Integer> population = new Vector<Integer>();
```

Avec ces déclarations, pays est une liste vide (aucun élément):

```
pays.size(); // vaut 0
```

Ajout d'un nouvel élément

```
pays.add("France");
pays.add("Espagne");//pays.size(); vaut 2
```

Accéder à un élément de la liste, les indices commence à 0

```
String nation = pays.get(1);//nation vaut Espagne
```

Insérer un élément, supposons que nous souhaitons insérer entre la France et l'Espagne, c'est à dire à

l'indice 1 , l'Italie.

```
pays.add(1, ''Italie'');
```

L'Italie prend sa place à l'indice 1 et pousse l'Espagne à l'indice 2.

Supprimer un élément, supposons que nous souhaitons supprimer la France qui se trouve à l'indice 0.

```
pays.remove(0);
```

La taille du ArrayList vaut 2, on trouve à l'indice 0 l'Italie et à l'indice 1 l'Espagne .
La boucle for permet de parcourir une liste à l'aide des indices :

```
for(int i = 0; i < pays.size(); i++) {
        System.out.println(pays.get(i) );
    }
```

La syntaxe de l'instruction for peut être simplifiée comme pour les tableaux :

```
for(String nation : pays) {
        System.out.println(nation);
    }
```

Un ArrayList fournit un accès aux éléments par leur indice très performant et est optimisé pour des opérations d'ajout/suppression d'éléments en fin de liste. Pour des raisons de performances, il est donc préférable d'utiliser ces derniers.

Class ArrayList<E>

Constructor Summary	
ArrayList() Constructs an empty list with an initial capacity of ten.	
ArrayList(int initialCapacity) Constructs an empty list with the specified initial capacity.	

Method Summary	
boolean	**add**(E e) Appends the specified element to the end of this list.
void	**add**(int index, E element) Inserts the specified element at the specified position in this list.
void	**clear**() Removes all of the elements from this list.
E	**get**(int index) Returns the element at the specified position in this list.
boolean	**isEmpty**() Returns true if this list contains no elements.
E	**remove**(int index) Removes the element at the specified position in this list.
boolean	**remove**(Object o) Removes the first occurrence of the specified element from this list, if it is present.
E	**set**(int index, E element) Replaces the element at the specified position in this list with the specified element.
int	**size**() Returns the number of elements in this list.

HashMap

L'interface `java.util.Map` permet d'associer une clef à une valeur et de retrouver une clef à partir d'une valeur. Parmi les implémentations de cette interface, on trouve la classe HashMap (k,v) qui permet d'associer des clés de type K à des valeurs de types V .
Exemple, on souhaite stocker dans une même collection un couple nom (clé), numéro de téléphone (valeur)

```
Map<String, String> contacts = new HashMap<String, String>();
```

Pour ajouter des éléments :

```
contacts.put("durand", "06.12.13.14.16");
contacts.put("dupont", "06.42.32.14.16");
```

System.out.println(contacts); va afficher:
{dupont=06.42.32.14.16, duret=06.45.83.14.16, durand=06.12.13.14.16}

System.out.println(contacts.get("durand")); va afficher :
06.12.13.14.16

Pour afficher toutes les clés :

```
for(String nom : contacts.keySet()) {
            System.out.println(nom);
        }
```

Pour afficher toutes les valeurs :

```
for(String telephone : contacts.values()) {
            System.out.println(telephone);
        }
```

Depuis **Java 1.5**, on peut aussi utiliser la boucle **for étendu** pour parcourir une Map. Il suffit d'utiliser la méthode *entrySet()* qui renvoie une collection de type Map.Entry qui représentent un couple clé-valeur. Voici un exemple :

```
//Pour récupérer les clefs et les valeurs
for (Map.Entry<TypeClefs, TypeValeurs> entry : map.entrySet()){
      System.out.println(entry.getKey() + " : " + entry.getValue());
}
```

Pour notre exemple :

```
for (Map.Entry<String, String> entry : contacts.entrySet()) {
      System.out.println(entry.getKey() + " : " + entry.getValue());
}
```

Pour en savoir plus : http://java.developpez.com/faq/java/?page=langage_donnees#LANGAGE_COLLECTIONS_info_map

Notion d'interfaces

Si vous consultez l'API JAVA, vous verrez que List et Map ne sont pas des classes mais des interfaces. Dans Java, les interfaces sont un type particulier de classes, qui permettent à une autre classe d'hériter un comportement auquel elle n'aurait pas accès autrement. Elles définissent des ensembles de méthodes implémentées par les classes du programme.
Il faut savoir que la hiérarchisation Objet de Java fait qu'une classe ne peut hériter que d'une seule classe, et non plusieurs. Les interfaces sont donc un moyen de se construire un environnement réutilisable pour toutes les classes.
Surtout, les interfaces servent à créer des comportements génériques: si plusieurs classes doivent obéir à un comportement particulier, on créé une interface décrivant ce comportement. Nous reviendrons sur ces notions, si vous voulez plus d'informations : http://profs.vinci-melun.org/profs/okpu/cours/coursJava/index.html