Chapitre 11

Collectionner un nombre indéterminé d'objets

- Les tableaux permettent la manipulation rapide et efficace d'un ensemble de données, mais leur taille est fixe.

- Java propose des classes qui permettent de stocker dynamiquement un ensemble d'objets :

  - **`Vector`**
  - **`LinkedList`**
  - **`Stack`**
  - **`Hashtable`**
  - **`ArrayList`**
  - **`...`**

1. Le type `Vector` (vecteur)

- Sa gestion ressemble à celle d'un tableau
  - ⇒ création d'une liste par ajout de données au fur et à mesure des besoins de l'utilisateur
  - ⇒ les données sont enregistrées dans leur ordre d'arrivée
  - ⇒ gestion automatique de l'indice permettant de retrouver les données
- Manipulation d'un vecteur

```
Vector<objet> liste = new Vector<objet>();
```

  - On déclare ainsi un objet de type **`Vector`** qui contient des éléments de type **`objet`**. Par exemple :

```
Vector<String> liste = new Vector<String>();
```

  - Utilisation des méthodes de la classe **`Vector :`**

- Les méthodes de la classe Vector

  - **`void add (indice, objet)`**

    Insérer un élément objet dans la liste à l'indice spécifié en paramètre, les éléments qui suivaient sont décalés.

  - **`void addElement (object)`**

    Ajouter un élément objet en fin de liste.

  - **`Object elementAt(indice)`**

    Retourner l'élément stocké à l'indice spécifié en paramètre.

  - **`int indexOf(objet)`**

    Retourner l'indice de la première occurrence de l'*objet* donné en paramètre, ou -1 si objet n'existe pas dans la liste.

- **`void clear()`**

  Supprimer tous les éléments de la liste.

- **`Object remove (indice)`**

  Supprimer l'objet dont l'indice est spécifié en paramètre.

- **`void removeRange(i,j)`**

  Supprimer tous les éléments compris entre les indices **`i`** (valeur comprise) et **`j`** (valeur non comprise).

- **`void setElementAt(Object, i)`**

  Remplacer l'élément situé en position **`i`** par l'objet spécifié en paramètre.

- **`int size()`**

  Retourner le nombre d'éléments placés dans la liste.

- Exemple d'utilisation : gestion d'un nombre indéterminé d'étudiants

Fichier Etudiant.java :

```
public class Etudiant
{
   private String nom;
   private double [] notes;
   private double moyenne;
```

```
// Constructeur

public Etudiant()
{
     Scanner clavier = new Scanner(System.in);

     System.out.print("Entrer le nom de l'etudiant : ");
     nom = clavier.nextLine();
     System.out.print("Combien de notes pour l'etudiant");
     System.out.print(nom + " :  ");
     int nombre = clavier.nextInt();
     notes = new double [nombre];
     for (int i = 0; i < notes.length; i ++)
     {
       System.out.print("Entrer la note  n°"+ (i+1)+ ":");
       notes[i] = clavier.nextDouble();
     }
     moyenne = calculMoyenne();
}
```

```
   public void afficheUnEtudiant()
   {
      System.out.print("Les notes de "+nom+ " sont : ");
     for (int i = 0; i < notes.length; i ++)
         System.out.print(" "+notes[i]);
     System.out.println("Sa moyenne vaut  "+ moyenne);
   }
}
```

Fin du fichier Etudiant.java

Fichier Classe_etudiants.java :

```
import java.util.*;

public class Classe_etudiants
{
   private Vector<Etudiant> liste;

   public Classe_etudiants()
   {
      liste = new Vector<Etudiant>();
   }

   public void ajouteUnEtudiant()
   {
       Etudiant  un_etudiant = new Etudiant();
       liste.addElement(un_etudiant);
   }
```

```
    public void afficheLesEtudiants()
    {
       int nbEtudiants = liste.size();
       if (nbEtudiants > 0)
      {
           Etudiant tmp;
           for (int i = 0; i < nbEtudiants; i ++)
           {
               tmp = liste.elementAt(i);
               tmp.afficheUnEtudiant();
            }
       }
       else
           System.out.println("Il n'y a pas d'etudiant
                                  dans cette liste");
     }
}
```

Fin du fichier Classe_etudiants.java

- Pour utiliser la classe **`Vector`** il faut faire appel à une librairie supplémentaire (**package**)

  $\Rightarrow$ utilisation de l'instruction **`import`** en première ligne du fichier qui utilise l'outil souhaité

```
import java.util.*
```

- Utilisation des méthodes suivantes

  - Le constructeur **`Classe_etudiants()`** :
    - il fait appel au constructeur de **`Vector`** afin de créer la liste.

  - La méthode **`ajouteUnEtudiant()`** :
    - place un élément dans la liste grâce à la méthode **`addElement()`**
    - L'élément ajouté est un objet de type **`Etudiant`**, créé par l'intermédiaire du constructeur **`Etudiant()`**
    - La taille de la liste est automatiquement augmentée.
    - Remarque : l'ajout d'un élément dans un vecteur n'est possible que si l'élément est un **objet**.

– La méthode **`afficheLesEtudiants()`** :

  - Parcourt l'ensemble de la liste grâce à la méthode **`elementAt()`**, qui fournit en résultat une référence sur l'élément stocké à la position spécifiée en paramètre.

Fichier GestionClasse.java :

```
public class GestionClasse
{
   public static void main(String [] args)
   {
      byte choix = 0 ;
      Classe_etudiants C = new Classe_etudiants();
      Scanner clavier = new Scanner(System.in);

      do
      {
         System.out.println("1. Ajoute un etudiant");
         System.out.println("2. Affiche la classe");
         System.out.println("3. Pour sortir");
         System.out.print("Votre choix : ");
         choix  = clavier.nextByte();
```

```
      switch (choix)
      {
         case 1 :
            C.ajouteUnEtudiant();
            break;
         case 2 :
            C.afficheLesEtudiants();
            break;
         case 3 :
            System.exit(0);
         default :
            System.out.println("option inexistante");
      }
    } while (choix != 3);
  }
}
```

Fin du fichier GestionClasse.java

2. Le type `Stack` (pile)

2.1 Présentation

- Gestion d'une pile contenant des objets.
- Principe : accès uniquement au dernier élément ajouté.
- Les objets empilés ou dépilés sont des références.
- **`Stack`** est une sous-classe de **`Vector`**.

  → on peut utiliser toutes les méthodes de **`Vector`**.

2.2 Les méthodes de la classe `Stack`

- **`boolean empty()`**

  Renvoie true si la pile est vide.

- **`Object peek()`**

  Renvoie le premier objet disponible sur la pile, mais sans le dépiler.

- **`Object pop()`**

  Dépile le premier objet disponible sur la pile.

- **`Object push(Object obj)`**

  Empile l'objet **`obj`** et le retourne.

2.3 Remarques

- La classe **`Stack`** a un seul constructeur sans argument :

```
Stack<objet> pile = new Stack<objet>();
```

  Par exemple :

```
Stack<String> pile = new Stack<String>();
```

- La classe **`Stack`** étant basée sur la classe **`Vector`**, le premier objet disponible, au sommet de la pile, est en fait le dernier du vecteur.

3. Le type `ArrayList`

https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/util/ArrayList.html

Classe permettant de définir des tableaux redimensionnables. Cette classe générique permet de stocker des objets, et pas de types simples (int, float, boolean, etc.).

```
import java.util.ArrayList;
```

```
ArrayList<String> noms = new ArrayList<String>();
```

Si vous voulez utiliser une **ArrayList** de types simples, il faut utiliser des classes enveloppes (voir chapitre 14), qui sont des classes qui encapsulent les types simples : **Integer** pour int, **Float** pour float, **Double** pour double, **Boolean** pour boolean, **Character** pour char.

```
import java.util.ArrayList;

public class MaClasse {
  public static void main(String[] args) {
    ArrayList<Integer> nombres = new ArrayList<Integer>();
    nombres.add(10);
    nombres.add(15);
    nombres.add(20);
    nombres.add(25);
    for (int i : nombres) {
      System.out.println(i);
    }
  }
}
```

Différences entre **ArrayList** et **Vector** :

ArrayList	Vector
ArrayList n'est pas synchronisé, ce qui peut poser problème dans un environnement multithread	Vector est synchronisé : dans un environnement concurent, il va bloquer l'accès au Vector aux autres processus jusqu'à que le premier thread autorise l'accès
ArrayList incrémente sa taille actuelle de 50% si le nombre des éléments dépassent sa capacité	Vector incrémente sa taille de 100%, si le nombre des éléments dépassent sa capacité
ArrayList est rapide parce qu'elle n'est pas synchronisé	Vector est lent parce qu'il est synchronisé

1) Ajout d'élément à une `ArrayList`

On ajoute un élément en fin d'une **ArrayList** avec la méthode **add()**.

```
import java.util.ArrayList;

public class MaClasse {
  public static void main(String[] args) {
    ArrayList<String> marques = new ArrayList<String>();
    marques.add("Volvo");
    marques.add("BMW");
    marques.add("Renault");
    marques.add("Mazda");
  }
}
```

2) Lire un élément d'une `ArrayList`

On lit un élément d'une **ArrayList** avec la méthode **get()**.

```
import java.util.ArrayList;

public class MaClasse {
  public static void main(String[] args) {
    ArrayList<String> marques = new ArrayList<String>();
    marques.add("Volvo");
    marques.add("BMW");
    marques.add("Renault");
    marques.add("Mazda");
    System.out.println(marques.get(0));        // --> Volvo
  }
}
```

3) Modifier un élément d'une `ArrayList`

On modifie un élément d'une **ArrayList** avec la méthode **set()**.

```
import java.util.ArrayList;

public class MaClasse {
  public static void main(String[] args) {
    ArrayList<String> marques = new ArrayList<String>();
    marques.add("Volvo");
    marques.add("BMW");
    marques.add("Renault");
    marques.add("Mazda");
    marques.set(0, "Opel");
    System.out.println(marques);
  }
}
```

4) Supprimer un élément d'une `ArrayList`

On supprime un élément d'une **ArrayList** avec la méthode **remove()**.

```
import java.util.ArrayList;

public class MaClasse {
  public static void main(String[] args) {
    ArrayList<String> marques = new ArrayList<String>();
    marques.add("Volvo");
    marques.add("BMW");
    marques.add("Renault");
    marques.add("Mazda");
    marques.remove(0);
    System.out.println(marques);
  }
}
```

Vider une **ArrayList** : **clear()**.

```
ArrayList<String> liste = new ArrayList<String>();
liste.add("Peugeot");
liste.add("Renault");
liste.clear();
```

Connaître le nombre d'éléments d'une **ArrayList** : **size()**.

```
ArrayList<String> liste = new ArrayList<String>();
liste.add("Peugeot");
liste.add("Renault");
System.out.println(liste.size());       // --> 2
```

Parcourir une **ArrayList** avec une boucle **for** :

```
public class MaClasse
{
  public static void main(String[] args)
  {
    ArrayList<String> liste = new ArrayList<String>();

    liste.add("Volvo");
    liste.add("BMW");
    liste.add("Ford");
    liste.add("Mazda");

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

Parcourir une **ArrayList** avec une boucle **for-each** :

```
public class MaClasse
{
  public static void main(String[] args)
  {
    ArrayList<String> liste = new ArrayList<String>();

    liste.add("Volvo");
    liste.add("BMW");
    liste.add("Ford");
    liste.add("Mazda");

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

4. Le type `Hashtable` (dictionnaire)

4.1 Présentation

- Améliorer la recherche d'éléments dans une liste

  $\Rightarrow$ associer une clé unique à chaque élément.

- Utilisation du type **`Hashtable`** pour réaliser l'association clé-élément.

- Syntaxe pour déclarer un dictionnaire :

```
Hashtable<k,v> listeClassée = new Hashtable<k,v>();
```

Exemple :

```
Hashtable<String,Integer> listeClassée = new
Hashtable<String,Integer>();
```

4.2 Méthodes de la classe `Hashtable`

- Object put (Object clé, Object élément)

  Place dans le dictionnaire l'association clé-élément.

- Object get (Object clé)

  Retourne l'objet associé à la clé spécifiée en paramètre.

- remove(Object clé)

  Supprime dans le dictionnaire l'association clé-objet à partir de la clé spécifiée en paramètre.

- int size()

  Retourne le nombre d'associations définies dans le dictionnaire.

4.3 Exemple

Création d'un dictionnaire d'étudiants.

On va insérer des objets de type **`Etudiant`** dans le dictionnaire, en associant une clé à chaque objet.

1) Définir une clé d'association

- Identifions un étudiant par son nom et son prénom
- Clé = chaîne de caractères en majuscule, composée :
  - du premier caractère du prénom
  - des caractères du nom

```
private String créerUneClé (Etudiant e)
{
  String tmp;
  tmp = (e.prénom).charAt(0)+ e.nom;
  tmp.toUpperCase();
  return tmp;
}
```

2) création des deux méthodes `quelPrénom()` *et* `quelNom()`

Dans le fichier Etudiant.java, ajouter :

```
public String quelNom()
{
    return nom;
}

public String quelPrénom()
{
    return prénom;
}
```

3) Création du dictionnaire

- Créer un objet de type **`Hashtable`**
- Stocker les étudiants dans cet objet en les associant à leur clé.

```
import java.util.*;

public class Classe
{
   private Hashtable<String,Etudiant>  listeClassée;
   public Classe()
   {
      listeClassée = new Hashtable<String,Etudiant>();

   }

   public void ajouteUnEtudiant()
   {
      Etudiant nouveau = new Etudiant();
      String clé = créerUneClé(nouveau);
      if (listeClassée.get(clé) == null)
         listeClassée.put(clé, nouveau);
      else
         System.out.println("Étudiant déjà saisi!");
   }
...
```

- Le constructeur **`Classe()`** fait appel au constructeur de la classe **`Hashtable`** afin de déterminer l'adresse du premier élément de **`listeClassée`**

- **`ajouteUnEtudiant()`** : place un élément du dictionnaire grâce à la méthode **`put(clé,nouveau)`**

- **`put(clé,nouveau)`** : ajoute l'association clé-nouveau dans le dictionnaire **`listeClassée`**

- L'ajout successif de deux associations ayant la même clé
  → destruction de la première association
  → tester que la clé n'existe pas !

4) Rechercher un élément du dictionnaire

- On ajoute à la classe **`Classe`** la méthode suivante :

```
public void rechercheUnEtudiant(String p,String n)
{
   String clé = créerUneClé(p, n);
   Etudiant eClassé = listeClassée.get(clé);
   if (eClassé != null)
      eClassé.afficheUnEtudiant();
   else
      System.out.println("étudiant inconnu!");
 }
```

5) Supprimer un élément du dictionnaire

- On ajoute à la classe **`Classe`** la méthode suivante :

```
public void supprimeUnEtudiant(String p, String n)
{
   String clé = créerUneClé(p, n);
   Etudiant eClassé = listeClassée.get(clé);
   if (eClassé != null)
   {
      listeClassée.remove(clé);
      System.out.println("Étudiant supprimé.");
   }
   else
      System.out.println("Étudiant inconnu !");
 }
```

6) Afficher un dictionnaire

- Il faut le parcourir élément par élément

- Utilisation d'un outil java défini par la classe **`Enumeration`**
  - Utilisation des méthodes :
    - **`hasMoreElements()`** : détermine s'il existe encore des éléments dans l'énumération.
    - **`nextElement()`** : permet l'accès à l'élément suivant dans l'énumération.

```
public void afficheLesEtudiants()
{
   if(listeClassée.size() != 0)
   {
      Enumeration enumEtudiant = listeClassée.keys();
      while (enumEtudiant.hasMoreElements())
      {
         String clé = (String)enumEtudiant.nextElement();
         Etudiant eClassé=listeClassée.get(clé);
         eClassé.afficheUnEtudiant();
      }
   }
   else
      System.out.println("Aucun étudiant dans la liste");
 }
```

- L'énumération est définie grâce à la méthode **`keys()`** de la classe **`Hashtable`**, qui renvoie sous forme d'énumération la liste des clés effectivement stockées.

- Le parcours de cette énumération est ensuite réalisée à l'aide d'une boucle while s'il existe encore des clés dans la liste (**`enumEtudiant.hasMoreElements()`**).

- Si c'est la cas on passe à l'élément suivant (**`enumEtudiant.nextElement()`**). Recherche de l'élément grâce à **`listeClassée.get(clé)`** et affichage grâce à **`eClassée.afficheUneEtudiant()`**.

5. Le type `LinkedList` (liste chaînée)

Ensemble ordonné d'éléments de même type auxquels on accède séquentiellement.

Classe de liste doublement chaînée présente dans le package **`java.util.LinkedList`**

Comme la classe Vector, les éléments de la classe **`LinkedList`** ne peuvent être que des objets et non des type élémentaires (byte, short, int, long ou char ne sont pas autorisés).

```
LinkedList<objet> liste = new LinkedList<objet>();
```

Exemple :

```
LinkedList<String> liste = new LinkedList<String>();
```

ajouter un élément au début de la liste

void addFirst(Object obj)

ajouter un élément à la fin de la liste

void addLast(Object obj)

effacer tous les éléments de la liste

void clear( )

élément situé au rang = 'index'

Object get(int index)

rang de l'élément 'elem'

int indexOf(Object elem)

efface l'élément situé au rang = 'index'

Object remove(int index)

remplace l'élément de rang 'index' par obj

Object set(int index , Object obj)

nombre d'éléments de la liste

int size( )

Exemple

```
import java.util.LinkedList;

class ApplicationLinkedList
{

 //affiche une liste de chaînes
 static void afficheLinkedList (LinkedList<String> liste)
 {
   System.out.println("taille= "+liste.size());
   for ( int i = 0; i < liste.size( ); i++ )
        System.out.println(liste.get(i));
 }
```

```
 static void initialiseLinkedList( )
 {
     LinkedList<String> liste = new LinkedList<String>( );
     for ( int i = 0 ; i < 5 ; i++ )
        liste.addLast( "val:" + String.valueOf(i));
     afficheLinkedList(liste);
 }

 static void main(String[] args)
 {
     initialiseLinkedList( );
 }
}
```

6. Tri de listes

Les classes Vector, ArrayList, LinkedList, peuvent être triées soit au moyen de l'API Collections, soit directement au moyen de leur méthode **sort()**.

Deux cas de figure :

1) Soit ces listes contiennent des éléments que Java peut comparer entre eux (String, Integer, Double, ...), et il suffit d'appeler directement la méthode **sort()** pour les trier ;

2) Soit ce sont des éléments que Java ne sait pas comparer (par exemple parce qu'il s'agit d'une classe avec plusieurs attributs que vous avez écrit, ex : classe Etudiant), pour que sort() puisse fonctionner il va falloir écrire une classe de comparaison qui implémente l'interface **Comparator**.

Exemple : tri ascendant d'une ArrayList de String

```
import java.util.*;

public class MaClasse {
  public static void main(String[] args) {
    ArrayList<String> liste = new ArrayList<String>();
    liste.add("Volvo");
    liste.add("BMW");
    liste.add("Ford");
    liste.add("Mazda");

    liste.sort();  // Tri alphabétique de la liste

    for (String i : liste) {
      System.out.println(i);
    // --> BMW, Ford, Mazda, Volvo
    }
  }
}
```

Exemple : tri descendant d'une ArrayList de String

```
import java.util.*;

public class MaClasse {
  public static void main(String[] args) {
    ArrayList<String> liste = new ArrayList<String>();
    liste.add("Volvo");
    liste.add("BMW");
    liste.add("Ford");
    liste.add("Mazda");

    liste.sort(Comparator.reverseOrder());

    for (String i : liste) {
      System.out.println(i);
    // --> Volvo, Mazda, Ford, BMW
    }
  }
}
```

Si on veut trier une liste contenant des objets complexes, il faut dire comment comparer deux de ces objets grâce à l'interface **Comparator**.

```
import java.util.*;
import java.lang.*;
import java.io.*;

class Etudiant
{
    String nom, prenom;
    double moyenne;

    public Etudiant(String nom, String prenom, double moyenne)
    {
        this.nom = nom;
        this.prenom = prenom;
        this.moyenne = moyenne;
    }

    public void affiche()
    {
        System.out.println( nom + " " + prenom + " : " + moyenne );
    }
}
```

```
class TriParMoyenne implements Comparator<Etudiant>
{
    public int compare(Etudiant a, Etudiant b)
    {
        if( a.moyenne < b.moyenne )
            return -1;
        else if( a.moyenne > b.moyenne )
            return 1;
        else
            return 0;
    }
}
```

```
class Principal
{
    public static void main (String[] args)
    {
        ArrayList<Etudiant> ar = new ArrayList<Etudiant>();
        ar.add(new Etudiant("Durand", "Jean", 12.5));
        ar.add(new Etudiant("Martin", "Alexandra", 15.7));
        ar.add(new Etudiant("Aubry", "Benoit", 9.5));

        ar.sort(new TriParMoyenne());

        for (int i=0; i<ar.size(); i++)
            ar.get(i).affiche();
    }
}
```