Collectionner un nombre indéterminé d'objets

Les tableaux permettent la manipulation rapide et efficace d'un ensemble de données: leur principal inconvénient est d'être de taille fixe.

Ainsi, l'ajout d'un élément dans un tableau demande une gestion rigoureuse des indices afin d'éviter que ces derniers ne prennent une valeur supérieure à la taille du tableau.

Pour pallier cette difficutié le langage Java propose plusieurs outils de manipulation des données en mémoire vive, au fur et à mesure des besoins de l'application.

En outre, lorsqu'un programme utilise des collections importantes de données, il doit les archiver de façon à ne pas les voir disparaître après l'arrêt de l'application ou de l'ordinateur.

En effet, sans programmation dynamique, vous pourriez voir une bibliothéque refuser de nouveaux lecteurs sous prétexte que le logiciel qu'elle utilise ne serait pas en mesure de traiter plus de 50 000 inscriptions.

La Classe `Vector`

```
Vector liste = new Vector() ;
```

• Ajoute un élément objet en fin de liste : `add(objet)`

• Insère un élément objet dans la liste, a l'indice spécifié en paramètre : `add(indice, objet)`

• Ajoute un élément objet en fin de liste et augmente sa taille de un. `addElement(objet)`

• Retourne l'élément stocké a l'indice spécifié en paramètre `elementAt(indice)`

• Supprime tous les éléments de la liste : `clear()`

• Retourne l'indice dans la liste du premier objet donné en paramètre : `indexOf(objet)`

• Retourne l'indice dans la liste du dernier objet donne en paramètre : `lastIndexOf()`

• Supprime l'objet dont l'indice est spécifié en paramètre : `remove(indice)`

• Supprime tous les éléments compris entre les indices i (valeur comprise) et j (valeur non comprise) : `removeRange(i,j)`

• Remplace l'élément situé en position i par l'objet spécifié en paramète : `setElementAt(objet,i)`

• Retourne le nombre d'éléments placés dans la liste : `size()`

Exemple

```
import java.util.*;
  public class Classe
  { private Vector liste;
    public Classe()
    { liste = new Vector();
    }
    public void ajouteUnEtudiant()
    { liste.addElement(new Etudiant());
    }
    public void afficheLesEtudiants()
    { int nbEtudiants = liste.size();
      if (nbEtudiants > 0)
      { Etudiant tmp;
        for (inf i = 0; i < nbEtudiants; i ++}
        {  tmp = (Etudiant) liste.elementAt(i);
           tmp.afficheUnEtudiant();
        }
      }
      else System.oui.printltln("II n'y a pas
                    d'etudiant dans cette liste")
     }
   } // Fin de Classe
```

Les outils comme `Vector` sont proposés par le langage Java. Ils sont définis à l'intérieur de le classes, qui ne sont pas, par défaut, directement accessibles par le compilateur. C'est pourquoi le programmeur doit préciser au compilateur où se situe la librairie du langage Java définissant l'outil utilisé (`package`): `import java.util.*;` en tête du fichier.

En effet, si cette instruction fait défaut, le compilateur détecte une erreur du type : `Class Vector not found`

Les trois méthodes suivantes sont définies à l'intérieur de celle-ci :

- Le constructeur `Classe()`, qui fait appel au constructeur de la classe `Vector` afin de déterminer l'adresse du premier élément de la liste.
- La méthode `ajouteUnEtudiant()`, qui place un élément dans la liste grâce à la méthode `addElement()`. L'élément ajouté à la liste est un objet de type `Etudiant`, créé par l'intermédiaire du constructeur `Etudiant ()`, qui demande la saisie au clavier des données caractéristiques de l'étudiant à construire. Cela fait, la taille de la liste est automatiquement augmentée de 1 par l'interpréteur.

- La méthode `afficheLesEtudiants()` parcourt l'ensemble de la liste grâce à la méthode `elementAt()`, qui fournit en résultat l'élément stocké à la position spécifiée en paramètre, soit i. Ce résultat, pour être consultable, doit obligatoirement être "caste" en `Etudiant`.

```
public class GestionClasse
{ public static void main(String [] argument)
  { byte choix = 0 ;
    Classe C = new Classe();
    do
    { System.eut.println("1. Ajoute un étud.");
      System.out.println("2. Affiche la classe");
      System.out.println("3. Pour sortir");
      System.out.print "Votre choix ; ");
      choix = Lire.i();
      switch (choix)
      {  case 1 : C.ajouteUnEtudiant();
                  break;
         case 2 : C.afficheLesEtudiants();
                  break;
         case 3 : System.exit(0);
         default : System.out.println("Cette
          option n'existe pas ");
      }
    } while (choix != 3);
  }
}
```

Les dictionnaires

Pour améliorer la recherche d'éléments complexes dans une liste, la technique consisie à organiser les données par rapport à une clé explicite. De cette façon, la recherche d'un objet dans la liste s'effectue, non plus sur l'ensemble des données qui le composent, mais sur une clé unique qui lui est associée.

L'organisation de données, par association d'une clé à un ensemble de données, est appelée un dictionnaire.

Dans un dictionnaire, chaque définition est associée au mot qu'elle définit et non pas à sa position (numérique) dans le dictionnaire.

Le type `Hashtable` proposé par le langage Java parmet de réaliser simplement l'association clé-élément. Les méthodes associées à ce type sont la création, la suppression, la consultation ou la modification d'une association. Pour utiliser un dictionnaire, il est nécessaire de la declare de façon suivante :

```
Hashtable listeClassée = new Hashtable () ;
```

Ainsi déclaré, `listeClassée` est un objet de type `Hashtable` sur lequel on peut appliquer des méthodes de la classe `Hashtable`. Les méthodes les plus couramment utilisées sont :

• Place dans le dictionnaire l'association clé-objet, `put(clé, objet)`

• Retourne l'objet sssocié à la clé spécifiée en paramètre `get(clé)`

• Supprime dans le dico. l'association clé-objet à partir de la clé spécifiée en paramètre `remove(clé)`

• Retourne le nombre d'associations définies dans le dictionnaire `size()`

Définir une clé d'association

En supposant qu'un étudiant soit totalement identifiable par son nom et son prénom, la clé d'association des données est définie comme étant une chaîne de majuscules, dont le premier caractère est le premier caractère du prénom de l'étudiant et les caractères suivants correspondent au nom de l'étudiant.

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

Les données `nom` et `prénom` de la classe `Etudiant` sont privées. Il est donc nécessaire d'uliliser les méthodes d'd'accès en consultation `quelPrénom()` et `quelNom( )` pour connaître le contenu de ces données.

Ces méthodes, à insérer dans le fichier `Etudiant.java`, sont décrites comme suit

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

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

La création d'une clé peut également être réalisée simplement à partir des `nom` et `prénom` de l'étudiant, stockés non pas dans un objet `Etudiant`, mais dans deux `String` distincts. Comment surcharger la méthode `créerUneclé( )` de façon à traiter cette alternative ?

```
private String créerUneClé(String p, String n)
{ String tmp;
  tmp = p.charAt(0)+ n;
  return tmp.toUpperCase( );
}
```

Création du dictionnaire

```
import java.util.*;
public class Classe
{ private Hashtable listeClassée;

  public Classe()
  { listeClassée = new Hashtable(); }

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

Ce programme est constitué des deux méthodes suivantes :

- Le constructeur `Classe()` qui fait appel au constructeur de la classe `Hashtable` afin de déterminer l'adresse du premier élémnt dans la liste;
- `ajouteUnEtudiant()`, qui place un élément dans le dico. grâce à `put(clé, nouveau)`, qui ajoute l'association clé-nouveau dans le dictionnaire `listeClassée`

L'ajout successif de deux associations ayant la même clé a pour résultat de détruire la première association. C'est pourquoi il convient de tester, avant de placer le nouvel étudiant dans le dictionnaire, si ce dernier n'est pas déjà dans la `listeClassée`. C'est ce que réalise

```
if (listeClassée.get(clé) == null)
  listeClassée.put(clé, nouveau);
```

Rechercher et supprimer un élément du dictionnaire

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

```
public void supprimeUnEtudiant
             (String p, String n)
{ String clé = créerUneClé(p, n);
  Etudiant eClassé =
           (Etudiant) listeClassée.get(clé);
  if(eClassé != null)
  { listeClassée.remove(clé) ;
     System.out.println(p + " " +
                        n + " a été supprime ");
  }
}
```

Afficher un dictionnaire

• `hasMoreEiements()` détermine s'il existe encore des éléments;

• `nextElement()` permet l'accès à l'élément suivant

• classe `Enumeration`

```
public void afficheLesEtudiants()
{  if(listeClassée.size() != 0)
   { Enumeration enumEtudiant =
                              listeClassée.keys();
     while ( enumEtudiant.hasMoreElements())
    { (String) enumEtudiant.nextEtudiant();
       Etudiant eClassé =
              (Etudiant) listeClassée.get(clé);
       eClassée.afficheUnEtudiant();
    }
   }
   else
   System.out.println("II n'y a
        pas d'etudiant dans cette liste");
}
```

Les dictionnaires

Pour améliorer la recherche d'éléments complexes dans une liste, la technique consisie à organiser les données par rapport à une clé explicite. De cette façon, la recherche d'un objet dans la liste s'effectue, non plus sur l'ensemble des données qui le composent, mais sur une clé unique qui lui est associée.

L'organisation de données, par association d'une clé à un ensemble de données, est appelée un dictionnaire.

Dans un dictionnaire, chaque définition est associée au mot qu'elle définit et non pas à sa position (numérique) dans le dictionnaire.

Le type `Hashtable` proposé par le langage Java parmet de réaliser simplement l'association clé-élément. Les méthodes associées à ce type sont la création, la suppression, la consultation ou la modification d'une association. Pour utiliser un dictionnaire, il est nécessaire de la declare de façon suivante :

```
Hashtable listeClassée = new Hashtable () ;
```

Ainsi déclaré, `listeClassée` est un objet de type `Hashtable` sur lequel on peut appliquer des méthodes de la classe `Hashtable`. Les méthodes les plus couramment utilisées sont :

• Place dans le dictionnaire l'association clé-objet, `put(clé, objet)`

• Retourne l'objet sssocié à la clé spécifiée en paramètre `get(clé)`

• Supprime dans le dico. l'association clé-objet à partir de la clé spécifiée en paramètre `remove(clé)`

• Retourne le nombre d'associations définies dans le dictionnaire `size()`

Définir une clé d'association

En supposant qu'un étudiant soit totalement identifiable par son nom et son prénom, la clé d'association des données est définie comme étant une chaîne de majuscules, dont le premier caractère est le premier caractère du prénom de l'étudiant et les caractères suivants correspondent au nom de l'étudiant.

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

Les données `nom` et `prénom` de la classe `Etudiant` sont privées. Il est donc nécessaire d'uliliser les méthodes d'd'accès en consultation `quelPrénom()` et `quelNom( )` pour connaître le contenu de ces données.

Ces méthodes, à insérer dans le fichier `Etudiant.java`, sont décrites comme suit

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

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

La création d'une clé peut également être réalisée simplement à partir des `nom` et `prénom` de l'étudiant, stockés non pas dans un objet `Etudiant`, mais dans deux `String` distincts. Comment surcharger la méthode `créerUneclé( )` de façon à traiter cette alternative ?

```
private String créerUneClé(String p, String n)
{ String tmp;
  tmp = p.charAt(0)+ n;
  return tmp.toUpperCase( );
}
```

Création du dictionnaire

```
import java.util.*;
public class Classe
{ private Hashtable listeClassée;

  public Classe()
  { listeClassée = new Hashtable(); }

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

Ce programme est constitué des deux méthodes suivantes :

- Le constructeur `Classe()` qui fait appel au constructeur de la classe `Hashtable` afin de déterminer l'adresse du premier élémnt dans la liste;
- `ajouteUnEtudiant()`, qui place un élément dans le dico. grâce à `put(clé, nouveau)`, qui ajoute l'association clé-nouveau dans le dictionnaire `listeClassée`

L'ajout successif de deux associations ayant la même clé a pour résultat de détruire la première association. C'est pourquoi il convient de tester, avant de placer le nouvel étudiant dans le dictionnaire, si ce dernier n'est pas déjà dans la `listeClassée`. C'est ce que réalise

```
if (listeClassée.get(clé) == null)
  listeClassée.put(clé, nouveau);
```

Rechercher et supprimer un élément du dictionnaire

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

```
public void supprimeUnEtudiant
             (String p, String n)
{ String clé = créerUneClé(p, n);
  Etudiant eClassé =
           (Etudiant) listeClassée.get(clé);
  if(eClassé != null)
  { listeClassée.remove(clé) ;
     System.out.println(p + " " +
                        n + " a été supprime ");
  }
}
```

Afficher un dictionnaire

• `hasMoreEiements()` détermine s'il existe encore des éléments;

• `nextElement()` permet l'accès à l'élément suivant

• classe `Enumeration`

```
public void afficheLesEtudiants()
{  if(listeClassée.size() != 0)
   { Enumeration enumEtudiant =
                              listeClassée.keys();
     while ( enumEtudiant.hasMoreElements())
    { (String) enumEtudiant.nextEtudiant();
       Etudiant eClassé =
              (Etudiant) listeClassée.get(clé);
       eClassée.afficheUnEtudiant();
    }
   }
   else
   System.out.println("II n'y a
        pas d'etudiant dans cette liste");
}
```

Exceptions : un mécanisme de gestion des erreurs

Une opération qui produit une erreur (par exemple division par 0) lève une *exception*, qui se propage dans la pile des méthodes ayant conduit à l'erreur jusqu'à ce qu'elle soit *attrapée*: si elle n'est pas attrapée avant d'être propagée par la méthode `main`, elle provoque l'arrêt de l'execution.

```
try { bloc }
catch( UneException ){ que faire }
catch( UneAutre )    { que faire }
```

```
class Divise
{  static int div(int i, int j)
   { return i/j; }
   public static void main(String [] args)
   { int i=3, j=0, d=0;
     try { d=div(i,j);
         } catch (ArithmeticException e)
     { .print("division par zero");
     }
   }
}


try { i=parseInt(s);
    }
    catch(NumberFormatException e)
    { .print("s n'est pas un entier");
      .print("je prends i=0");
      i=0;
    }
```