Structures de données

Introduction

Les structures de données sont des conteneurs à données. Un des premiers chapitres de ce cours traite des matrices (ou tableaux) qui sont les structures de données les plus simples en Java (hormis les types de données primitifs). Les structures de données servent donc à stocker de manière ordonnée (selon une structure) des données (le plus souvent des objets Java). Les structures de données présentées dans ce chapitre sont des structures plus évoluées que les matrices.

Liste des structures

Toutes les structures de données de Java sont importables à partir du package java.util.

- BitSet
- Vector
- Stack
- Dictionary
- Hashtable
- L'interface Enumeration (qui n'est pas une réelle structure de données)

BitSet

La classe BitSet représente de jeux de bits, utile pour représenter un groupe de *flags* booleens. Elle adapte sa taille automatiquement au nombre de bits nécessaires.
Si vous débutez en Java, il est très improbable que vous ayez à l'utiliser ; aucun exemple de son utilisation ne sera donc donné ici.

Vector

C'est sûrement la structure de données que vous utiliserez le plus souvent. Un « vector » est simplement une matrice « dynamique », c'est à dire que l'on peut lui ajouter ou lui enlever des éléments après son initialisation.
Voici quelques méthodes utiles :

```
Vector a = new Vector() ; // Vector vide, initialisé "à vide".
Vector b = new Vector(int n); // Vector avec une contenance n.
Vector c = new Vector(int n, int m); // Vector avec une contenance de n et une
// progression de m (si m=5, le premier élément est en 0, le 2° en 5, le 3° en 10 ème position…).

// Ajouter des éléments sans spécifier de place.
 a.addElement("Je") ; a.addElement("Tu") ; a.addElement("Il") ;

// Récupère la valeur du dernier élément de « a ». Convertion obligatoire.
String s = (String)a.lastElement() ;

// Récupère la valeur d'un élément à la place donnée :
String s = (String)a.elementAt(int n) ;

// Insère un élément à une place donnée:
a.insertElementAt("Nous", 4); a.insertElementAt("Vous", 5); a.insertElementAt("Ils", 6);

// Enlève tout les éléments
a.removeAllElements() ;
```

```
// Recherche un élément dans un vector sans que l'on sache sa place :
boolean yEstIl = a.contains("Il") ; // renvoie true
boolean yEstIl = a.contains("Lui") ; // renvoie false

// Cherche la position d'un élément :
int i = a.indexOf("Tu") ; // i = 2

// Enlève un élément:
a.removeElement("Nous") ;

// Insère les éléments d'un Vector dans une Enumeration (voir plus loin)
Enumeration e = a.elements() ;

// Renvoie la taille d'un vector
int i = a.size() ; // i = 6

// Spécifie la taille d'un vector
a.setSize(10) ;
```

Stack

On peut se représenter une structure de données Stack sous la forme d'une pile de pièces de monnaie. Stack fonctionne avec le principe « premier entré, dernier sorti ». Dans la pratique Stack est très peu utilisé. Voici quelques méthodes utiles :

```
Stack s = new Stack() ;

// Ajouter un élément en haut de la pile
s.push("Je") ;
s.push("Tu") ;

// Supprimer des éléments :
String s = (String)s.pop() ;

// Atteindre l'élément en haut de la pile sans l'effacer :
String s = (String)s.peek() ;

// Retourne la distance qui sépare un élément du haut de la pile:
int i = s.search("Tu") ; // i = 1

// Détermine si une pile est vide:
boolean estVide = s.empty() ;
```

Dictionary

Une structure de données Dictionary se comporte comme une structure Vector, exception faite de l'accès aux données qui ne se fait pas par un indice mais par une "clé". Vous ne pouvez pas instancier Dictionary.

```
// Pour ajouter un élément à Dictionary :
dict.put("petit", new Rectangle(0, 0, 5, 5)) ;
dict.put("moyen", new Rectangle(5, 5, 10, 10)) ;
dict.put("grand", new Rectangle(10, 10, 15, 15)) ;

// Récupérer un élément:
Rectangle r = (Rectangle)dict.get("moyen") ;
```

```
// Récupérer la taille d'une structure Dictionary :
int i = dict.size();

// Savoir si une structure Dictionary est vide:
boolean estVide = dict.inEmpty();

// Convertir en "enumeration" les références aux objets ou les objets :
Enumeration keys = dick.keys() ;
Enumeration elements = dict.elements();
```

Hashtable

Hashtable est, comme Vector, une des structures de données les plus utilisées. C'est une classe abstraite dérivée de la classe Dictionary. En français, *table de hachage*.
Les capacités de stockage d'une Hashtable sont définies pas un *facteur de charge* compris entre 0.0 et 1.0. Comme les objets vector, les hashtable possèdent une taille (nombre d'éléments) et une capacité (mémoire utilisée). Une hashtable alloue la mémoire en considérant la taille actuelle de la table (de hachage) et en la comparant au produit (mathématique) du facteur charge par la capacité. Ainsi, plus le facteur de charge est élevé plus la gestion de la mémoire est efficace et plus la recherche d'un élément est lente. A l'inverse, quand le facteur de charge tend vers 0, la recherche d'un élément est rapide mais la gestion de la mémoire est mauvaise.

```
Hashtable a = new Hashtable() ;
Hashtable b = new Hashtable(int n); // n = capacité (et non taille) initiale spécifiée
Hashtable c = new Hashtable(int n, int m); // n = capacité, m = facteur de charge

// Les méthodes de la classe Hashtable sont les mêmes que celles de la classe Dictionary :
elements() ;
get() ;
inEmpty() ;
keys() ;
put();
remove();
size();

// Quelques méthodes propres à Hashtable:
// Enlève tous les éléments et clés :
a.clear() ;

// Recherche de la valeur d'un objet et non de sa clé :
boolean yEst = a.contains(new Rectangle(0, 0, 5, 5)) ;

// Quand une table de hachage doit reconsidérer sa capacité, elle se re-hache (!), on peut
// cependant forcer ce re-hachage sans augmenter la capacité de la table :
a.rehash() ;
```

En pratique, las tables de hachage serviront pour stocker des éléments complexes, car, comme le savent tous les programmeurs C/C++, il est plus rapide (facile ?) d'appeler la référence à un objet que l'objet lui même. Les clés des tables de hachage sont donc des pointeurs vers les éléments de la table. (Pour plus d'informations sur les pointeurs, voir le chapitre « Pointeur *this* et contexte statique »). Les clés calculées par une table de hachage sont des *codes de hachage*, en anglais Hash Code. En Java, le parent de toutes les classe dérivées de la classe Object définissent une méthode hashCode(). Toute classe qui définit une méthode hashCode() peut être stockée dans une table de hachage.

L'interface *enumeration*

Comme l'indique le titre du paragraphe, *enumeration* n'est pas une classe instanciable main une interface qui définit seulement 2 méthodes :

- public abstract boolean hasMoreElements() ;
- public abstract Object nextElement();

La première méthode définit s'il existe encore un élément après l'élément courant et la seconde récupère cet élément.
Voici une utilisation pratique :

```
while (monEnumeration.hasMoreElements()) {
    Object o = monEnumeration.nextElement();
    System.out.println(o);
}
```

Ceci affiche le contenu d'une enumeration.

L'interface enumeration est simple, pratique et efficace, vous l'utiliserez souvent. On peut utiliser enumeration en concert avec d'autres structures de données : exemple : des élements stockés dans une vecteur (classe Vector), exploitation de ces éléments suivant le schéma de la boucle while présentée ci-dessus.