Collections/Maps

Introduction :

- Les collections sont des objets permettant de gérer des ensembles d'objets avec éventuellement la possibilité de gérer les doublons, les ordres de tri, etc.

- Elles sont utilisées pour stocker, retrouver et manipuler des données, ainsi que pour transmettre des données d'une méthode à une autre.

- Les collections fournissent en plus une séparation entre leur implémentation effective et leur usage, permettant ainsi une meilleure réutilisabilité. Pour cela, on utilise l'héritage et les interfaces.

Exemples :
- un dossier de courrier : collection de mails
- un répertoire téléphonique : collection d'associations noms/numéros de téléphone.

La version 1 de Java proposait:
- `java.util.Vector, java.util.Stack, java.util.Hashtable`
- Une interface `java.util.iterator` permettant de parcourir ces objets.

Exemple de la collection "Stack"

```
import java.util.* ;
public class ExempleStack
{
   Stack pile ;
   public ExempleStack (){
      pile = new Stack () ;
      pile.push("Je suis ") ;
      pile.push("Un exemple ") ;
      pile.push("de pile");
      Iterator iter = pile.iterator () ;
      while (iter.hasNext()){
         System.out.println (iter.next()) ;
      }
   }
   public static void main(String[ ] args){
      new ExempleStack () ;
   }
}
```

Je suis
Un exemple
de pile

Exemple de la collection "TreeSet"

```
1 import java.util.*;
2
3  public class TestCollec {
4  public static void main(String argc[])
5  {
6      Set promotion= new TreeSet();
7      promotion.add(new Etudiant(1, "Turing"));
8      promotion.add(new Etudiant(2, "Babbage"));
9      if (promotion.contains(new Etudiant(4,"AAA")))
10        System.out.println("AAA appartient à la
          promotion");
11  }
12  }
```

*__ligne 1__ : les collections sont dans le package `java.util`

*__ligne 6__ : on crée une collection de classe `TreeSet`, et on la range dans un *handle* vers un `Set`.

*__ligne 7__ : On ajoute un nouvel étudiant dans l'ensemble « `promotion` », à l'aide de la méthode `add`.

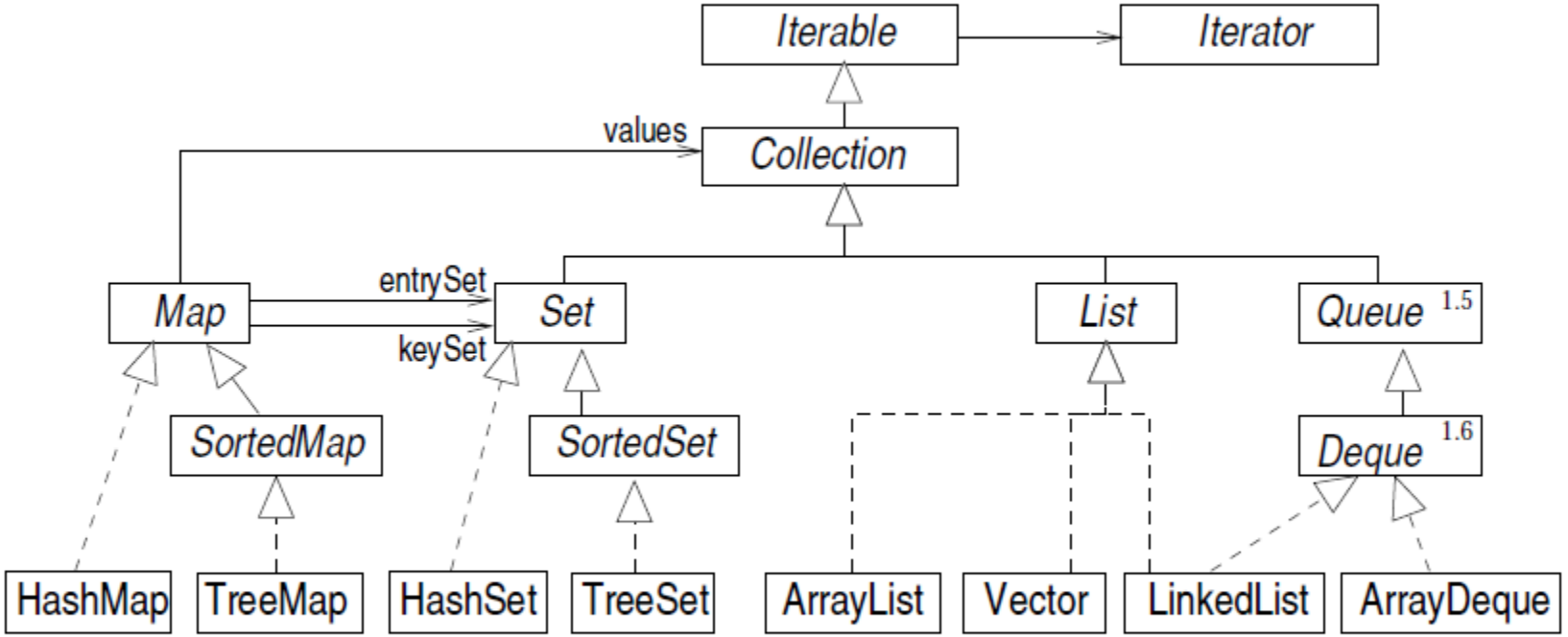
*__ligne 9__ : On teste si `AAA` appartient à la promotion, grâce à la méthode `contains`.

La ligne 6 est très importante
- Ce que l'on veut, dans la suite du programme, c'est un ensemble. Peu importe, pour la suite, la manière dont cet ensemble est implémenté.
- Par contre, lors de la création de cet ensemble, nous sommes bien forcé de choisir une implémentation effective.
- Nous choisissons ici `TreeSet` (implémentation d'un ensemble grâce à un arbre).

- Un point important est que nous ne précisons nulle part que `TreeSet` est un ensemble d'étudiants.
- En fait, `TreeSet` est un ensemble d'`Object` (pour ce type particulier d'ensemble, on a de plus l'obligation que les objets en question implémentent l'interface `Comparable`, c'est-à-dire qu'ils possèdent une méthode `compareTo`).

Collections à partir de java 2

Voici un extrait du diagramme de classes concernant les structures de données de Java.



Remarque : `Vector` est une « ancienne » classe qui a été modifiée pour faire partie de la hiérarchie `Collection`. C'est la seule classe « synchronisée ».

Collection : interface qui est implémentée par la plupart des objets qui gèrent des collections.

Set : interface pour des objets qui n'autorisent pas la gestion des doublons dans l'ensemble

SortedSet : interface qui étend l'interface Set et permet d'ordonner l'ensemble

List : interface pour des objets qui autorisent la gestion des doublons et un accès direct a un élément

Queue : interface qui étend l'interface Collection pour une file de données type FIFO

Dequeue : (double-ended queue) interface qui étend l'interface Collection pour une file de données type FIFO/LIFO

Map : interface qui définit des méthodes pour des objets qui gérent des collections sous la forme clé/valeur. Attention, ce n'est pas un sous-type de collection !

SortedMap : interface qui étend l'interface Map et permet d'ordonner l'ensemble. Un tableau associatif avec une relation d'ordre sur les clés.

ConcurrentMap : interface qui étend l'interface Map pour une association à accès concurrent

http://docs.oracle.com/javase/7/docs/api/java/util/Collection.html

http://docs.oracle.com/javase/tutorial/collections/interfaces/index.html

Collection<E> et types paramètrés

* Toutes les collections sont des types paramétrés par le type des éléments (E) qu'elles contiennent
* Les collections sont des conteneurs homogènes
* Si l'on veut stocker des éléments de types différents, on utilise le super-type commun, voir Object

Propriétés des collections

Sauf exceptions :
- Toutes les collections acceptent `null` comme un élément valide (mais ce n'est pas une bonne idée !)
- Toutes les collections testent si un objet existe ou non par rapport à la méthode `equals()` de l'objet
- Toutes les collections ne permettent pas les accès concurrents

Opérations optionnelles

- Certaines méthodes des collections sont des opérations optionnelles (signalées par un * - exemple : `boolean add*(E e)`)
- Celles-ci peuvent ne pas être implantées par exemple pour définir des collections immutables – exemple : `ImmutableList`
- Les opérations optionnelles non implantées lèvent l'exception `UnsupportedOperationException`

Les types génériques

Les types génériques permettent de spécifier le type d'objets que l'on va placer dans une collection d'objets (List, Vector,...)

Avantages:

* meilleure lisibilité: on connaît à la lecture du programme quel type d'objets seront placés dans la collection.
* La vérification peut être faite a la compilation.
* Le *cast* pour récupérer un objet de la collection est devenu implicite (sans cette fonctionnalité, il fallait faire un *cast* explicite, sachant que celui-ci peut échouer mais cela n'était détectable qu'a l'exécution)

La syntaxe pour utiliser les types génériques utilise les symboles < et >.

La généricité

* Une méthode peut être paramétrée avec des valeurs.

* La généricité permet de paramétrer du code avec des types de données.

Exemple :

```
class ArrayList<T>
```

* Ici le code est paramétré par un type T

* Pour l'utiliser il faut passer un type en argument :

```
new ArrayList<Employe>()
```

L'interface `java.util.Collection`

– `Collection` : super-type des structures de données de Java (sauf Map)

– Avant Java 1.5, les éléments d'une collection étaient du type `Object`
Depuis 1.5, les collections sont paramétrées par E, type des éléments

– Une collection peut autoriser ou non plusieurs occurrences d'un même élément (`List vs Set`)

– Les éléments peuvent être ordonnés (position) ou non (`List vs Set`)

– Les éléments peuvent être triés ou non (`Set vs SortedSet`)

– Pour limiter le nombre d'interfaces, certaines méthodes peuvent :
*ne pas être définies sur un sous-type (`UnsupportedOperationException`)
*lever `ClassCastException` (cf `checkedList`, etc.)

– Toute réalisation d'une `Collection` devrait définir :
* un constructeur par défaut (qui crée une collection vide) et
* un constructeur qui prend en paramètre une collection (conversion)

L'interface `java.util.Collection<E>`

```
public interface Collection<E> extends Iterable<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    // optional
    boolean add(E element);
    // optional
    boolean remove(Object element);
    Iterator<E> iterator();
    // Bulk operations
    boolean containsAll(Collection<?> c);
    // optional
    boolean addAll(Collection<? extends E> c);
    // optional
    boolean removeAll(Collection<?> c);
    // optional
    boolean retainAll(Collection<?> c);
    // optional
    void clear();
    // Array operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```


http://docs.oracle.com/javase/tutorial/collections/interfaces/collection.html

Taille et effacement

* L'interface définit les méthodes :
  - `int size()` indiquant la taille d'une collection
  - `boolean isEmpty()` indiquant si une collection est vide.
  - `void clear*()` permettant d'effacer les données de la collection

Modification et test de contenue

* Les modifications et tests sont effectuées par :
  - `boolean add*(E e)` ajoute un élément à la collection, *true* si la collection est modifiée
  - `boolean remove*(Object o)` retire un objet, *true* si la collection est modifiée
  - `boolean contains(Object)` renvoie si la collection contient l'objet

* `remove()` et `contains()` prennent en paramètre des `Object` et non des `E` par compatibilité

Éléments et equals

Toutes les collections sauf exception (`AbstractCollection` par exemple) testent si un objet existe en utilisant la méthode `equals()` de l'objet

```
public class MyPoint {
  public MyPoint(int x,int y) {
    this.x=x;
    this.y=y;
  }                                  // ne définit pas equals()
  private final int x,y;
  public static void main(String[] args) {
    Collection<MyPoint> c=new ArrayList<MyPoint>();
    c.add(new MyPoint(1,2));       // true
    c.contains(new MyPoint(1,2)); // false
  }
}
```

Iterator

Supposons que nous ayons rangé des étudiants dans un tableau. L'affichage de l'ensemble des étudiants se ferait de la manière suivante :

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

Supposons que nous ayons créé un type « liste d'étudiants ». Nous le parcourrions de la manière suivante :

```
for (ListeEtud l= listeEtuds; l != null; l= l.next)
   System.out.println(l.getEtud());
```

Du point de vue de l'implémentation, ces deux approches sont très différentes. Par contre, du point de vue de l'utilisation, on remarque que l et i sont similaires. Ils disposent chacun des fonctionnalités suivantes :

– on les initialise au « début » de l'ensemble `(i=0, l= listeEtuds)` ;
– on dispose d'un opérateur pour passer à l'élément suivant : `(i++ et l= l.next)` ;
– on dispose d'un test d'arrêt : `i < tabEtuds.length et l != null` ;
– on dispose d'une opération pour récupérer l'élément courant : `tabEtuds[i]` et `l.getEtud()`.

suite ...

De plus, ces quatre opérations permettent conceptuellement de parcourir n'importe quel ensemble.
En java, ce concept est actualisé par l'interface `Iterator`. Le code précédent peut alors s'écrire, pour n'importe quelle collection, de la manière suivante :

```
for (Iterator i= promotion.iterator(); i.hasNext(); )
{
   Etudiant e= (Etudiant) i.next();
   System.out.println(e);
}
```

Remarquez que c'est promotion qui construit l'itérateur. C'est encore un usage de l'héritage !

Les opérations vues précédemment étant représentées comme suit :
Création, initialisation La création d'un itérateur est effectuée par la collection elle-même, par l'appel de la méthode `iterator()` ;
Récupération de l'élément courant et avancement La méthode `next()` renvoie la valeur de l'élément courant, puis avance d'un cran (cf. i++ en C).
Test d'arrêt La méthode `hasNext` retourne vrai s'il est possible d'appeler `next()`

Pour résumer

```
public static void main(String[] args) {
  Collection<Integer> c=new ArrayList<Integer>();
  c.add(3);
  c.add(2);
  c.add(4);
  Iterator<Integer> it=c.iterator();
  for(;it.hasNext();) {
    System.out.println(it.next()*2);
  } // affiche 6, 4, 8
}
```

* Pour parcourir une collection, on utilise un objet permettant de passer en revue les différents éléments de la collection
* `java.util.Iterator<E>` :
  - `boolean hasNext()` qui renvoie vrai s'il y a un suivant
  - `E next()` qui renvoie l'élément courant et décale sur l'élément suivant
  - `void remove()` qui retire un élément précédemment envoyé par `next()`

next() et NoSuchElementException

L'opération `next()` est sécurisée et lève une exception dans le cas où on dépasse la fin de la collection (c-a-d si `hasNext()` renvoie `false`)

```
public static void main(String[] args) {
  Collection<Integer> c=new ArrayList<Integer>();
  c.add(3);
  Iterator<Integer> it=c.iterator();
  it.next();
  it.next(); // NoSuchElementException
}
```

Organisation des interfaces

L'organisation de l'arbre d'héritage pour les collections est le suivant :
(les interfaces définissent les grandes lignes de ce que peut faire une collection)

```
interface java.util.Collection
   interface java.util.List
   interface java.util.Queue
   interface java.util.Deque
   interface java.util.Set
      interface java.util.SortedSet
interface java.util.Map
   interface java.util.SortedMap
   interface java.util.ConcurrentMap
   .
   .
   .
```

Organisation des classes

```
class java.util.AbstractCollection (implements
java.util.Collection)
   class java.util.AbstractList (implements java.util.List)
      class java.util.AbstractSequentialList
         class java.util.LinkedList (implements java.util.List)
      class java.util.ArrayList (implements java.util.List)
      class java.util.Vector (implements java.util.List)
         class java.util.Stack
   class java.util.AbstractSet (implements java.util.Set)
      class java.util.HashSet (implements java.util.Set)
      class java.util.TreeSet (implements java.util.SortedSet)
   class java.util.AbstractMap (implements java.util.Map)
      class java.util.HashMap (implements java.util.Map)
      class java.util.TreeMap (implements java.util.SortedMap)
      class java.util.WeakHashMap (implements java.util.Map)
   .
   .
   .
```

Pour comprendre

– les listes (`List`) sont des suites d'éléments. On peut, soit les parcourir du premier au dernier, soit accéder au i ème élément ;

– les ensembles (`Set`) ne comportent pas de doublets. Un élément appartient à un ensemble ou ne lui appartient pas. Contrairement aux listes, on ne peux pas choisir où insérer un élément. Les `SortedSet` sont des ensembles triés, c'est-à-dire que les éléments seront rangés du plus petit au plus grand (selon la méthode `compareTo`).

– les dictionnaires (`Map`) sont des tableaux associatifs. C'est-à-dire des tableaux dont les indices sont d'un type quelconque. On peut par exemple avoir comme indice une chaîne de caractères, et écrire des choses comme :

```
definitions.put("chauve-souris", "mammifère insectivore volant");
```

– `Map.Entry`, permet de manipuler un dictionnaire comme un ensemble (`Set`) de dénitions.

En ce qui concerne les classes, elles fournissent des implémentations de ces interfaces. Le choix de la classe utilisée dépend de l'usage que l'on veut en faire.

– `LinkedList` fournit une liste doublement chaînée. L'ajout d'un élément est rapide, mais la recherche d'une valeur donnée et l'accès au ie élément sont en O(n).

– `ArrayList` est une implémentation à base de tableau. L'ajout d'un élément peut être plus coûteux que dans le cas d'une `LinkedList` si le tableau doit grossir, mais par contre l'accès au i ème élément est en temps constant.

– `Vector` est un équivalent de `ArrayList` datant du jdk1.0. Les différences entre les deux se situent surtout lorsque l'on utilise le multitâche (`Vector` est synchronisé, c.f. cours Threads).

– `Stack` est un type spécial de vecteur, utilisé pour réaliser des piles. Les piles sont des structures de données très utiles.

Pour le reste :

– `Hash` signifie que l'implémentation utilise un algorithme de hachage : elle **associe à chaque élément un nombre** (par exemple, on pourrait associer à une chaîne de caractères la somme des codes des caractères qu'elle contient), et **elle utilise ce nombre comme indice dans un tableau**. Les implémentations par hachage sont généralement très rapides pour des volumes moyens de données. Par contre, elles ne permettent pas de récupérer les éléments dans l'ordre.

– `Tree` signifie que l'implémentation utilise un arbre. Les arbres sont des structures très robustes, adaptées à de **grands volumes de données**. De plus, ils permettent de récupérer les éléments dans l'ordre croissant.

L'interface java.util.List<E>

Structure de données où chaque élément peut être identifié par sa position.

```
boolean add(E e)          // ajouter e à la fin de la liste
void add(int i, E e)      // ajouter e à l'index i
E set(int i, E o)
boolean addAll(int, Collection<? extend E> c)   // ... insérés à partir de i

E get(int i)              // élément à l'index i
int indexOf(Object o)     // index de l'objet o dans la liste ou -1
int lastIndexOf(Object o)       // dernier index de o dans la liste
List<E> subList(int from, int to)       // liste des éléments [from..to[

E remove(int i)           // supprimer l'élément à l'index i

ListIterator<E> listIterator()          // itérateur double sens
ListIterator<E> listIterator(int i)     // ... initialisé à l'index i

... et celles de collection !
```

La classe ArrayList<E>

Tout comme `LinkedList<E>` (liste chaînée), `ArrayList<E>` implante l'interface `List`. Elle est utiliser pour les tableaux à taille variable.

Constructeurs

- `ArrayList()`
- `ArrayList(int taille_initiale)` : peut être utile si on connaît la taille finale ou initiale (après les premiers ajouts) la plus probable car évite les opérations d'augmentation de la taille du tableau sous-jacent
- `ArrayList(Collection c)` : pour l'interopérabilité entre les différents types de collections

Méthodes principales

- `boolean add(E elt)`
- `void add(int indice, E elt)`
- `boolean contains(Object obj)`
- `E get(int indice)`
- `int indexOf(Object obj)`
- `Iterator<E> iterator()`
- `E remove(int indice)`
- `E set(int indice, E elt)`
- `int size()`

Exemple

```
List<Employe> le = new ArrayList<>();
Employe e = new Employe("Dupond");
le.add(e);
// Ajoute d'autres employés
//. . .
// Affiche les noms des employés
for (int i = 0; i < le.size(); i++) {
    System.out.println(le.get(i).getNom());
}
```

Pour aller plus vite :

```
for (Iterator i=le.iterator(); i.hasNext(); )
        i.next().getNom();
```

La classe java.util.Vector<E>

Objectif : Disposer de tableaux automatiquement redimensionnables.

Remarque : La classe Vector est définie dans le paquetage java.util.

Constructeurs : Sauf contre-indication, ils créent des vecteurs vides.

```
Vector(int capacitéInitiale, int incrémentCapactité)
Vector(int capacitéInitiale)    // Vector(capacitéInitiale, 4)
Vector()                        // Vector(10);
Vector(Collection<? extends E> c) // vecteur contenant les éléments de c
```

Méthodes élémentaires : (équivalentes à celles des tableaux)

```
int size()      // le nombre d'éléments du vecteur (taille effective)
int capacity()  // capacité du vecteur

E get(int index)        // vecteur[index]  0 <= index < size()
E elementAt(int index)  // get(index)

boolean add(E o)        // ajouter o comme dernier élément du vecteur avec
                        // redimensionnement si nécessaire (renvoie true)
boolean addElement(E o)  // idem add(Object)
add(int index, E o)      // vecteur[index] = o et 0 <= index <= size()
set(int index, E o)      // vecteur[index] = o et 0 <= index < size()
```

La classe java.util.Vector<E>, suite ...

```
E firstElement()          // get(0)
E lastElement()           // get(size()-1)

boolean isEmpty()         // aucun élément ?

void clear()              // supprimer tous les éléments du vecteur
void removeAllElements()        // idem clear()
boolean remove(Object o)        // supprime la première occurence
E remove(int index)       // supprimer vecteur[index] (changé ?)

boolean contains(Object o)      // vecteur contient-il o ?

int indexOf(Object o)           // premier index de o dans le vecteur
int indexOf(Object o, int i)    // ... à partir de l'indice i
int lastIndexOf(Object o)       // dernier index de o dans le vecteur
int lastIndexOf(Object o, int i)        // ... à partir de l'indice i

void setSize(int ns)      // changer la taille effective du vecteur

Object[] toArray()        // un tableau contenant les éléments du vecteur
```

Remarque : Il existe des méthodes qui manipulent une collection.

La classe java.util.Vector<E>, suite ...

• La manipulation des objets `Vector` est plus lente que celle des tableaux
• On ne peut y stocker que des objets (`Object`). Ils ne peuvent pas contenir des primitifs.
• L'insertion d'un objet entraîne un sur-casting implicite
• L'utilisation d'un élément du vecteur nécessite un sous-casting

Création d'un vecteur

- `Vector<Fraction> v ;`
- `v = new Vector<Fraction>();`
  - v est de taille 10 et d'incrément 1
  - l'incrément est l'augmentation de la taille du vecteur une fois plein
- `v = new Vector<Fraction>(int n);`
  - v est de taille n
- `v = new Vector<Fraction>(int n, int inc);`
  - v est de taille `n` et d'incrément `inc`
  - un incrément 0 double la taille du vecteur

Méthodes de Vector

- `v.size()` donne la taille du vecteur
- `v.setSize(int n)` modifie la taille de v
- `v.addElement (Object obj)` ajoute l'élément obj
- `v.setElementAt (Object obj,int n)` modifie l'élément n
- `Object obj = v.elementAt(int n)` extrait l'élément n

Exemple

```
import java.util.*;
class DesEntiers {
   static Vector<Integer> v;
   static Random r = new Random();
   public static void main (String [] args){
      v = new Vector<Integer>();
      for(int i =0; i<20;i++)
         v.addElement( new Integer(r.nextInt()));
      for(int i =0; i< v.size();i++){
         Integer obj = v.elementAt(i);
         System.out.println(""+ obj.intValue());
      }
   }
}
```

Exercices

• Modifier l'exemple précédent pour créer un vecteur de `Boolean` et `Integer` aléatoirement (le vecteur contient un mélange de ces objets) Imprimer les éléments du vecteur

• Créer un `Vector` de Voiture en créant aléatoirement le nombre et les attributs des objets Voiture

```
import java.util.*;
class DesEntiersEtBooléens {
    static Vector v;
    static Random r = new Random();
    public static void main (String [] args){
        v = new Vector(20, 0);
        int n = r.nextInt(30);
        for(int i =0; i<n;i++){
            if(r.nextBoolean())
                v.addElement( new Integer(r.nextInt()));
            else
                v.addElement( new Boolean(r.nextBoolean()));
        }
        for(int i =0; i<v.size();i++){
            Object obj = v.elementAt(i);
        if(obj.getClass().getName().equals("java.lang.Integer")) {
            System.out.println(""+ ((Integer)obj).intValue());
        }
        else {
            System.out.println(""+ ((Boolean)obj).booleanValue());
        }
        }
    }
  }
}
```

```
import java.util.*;
class DesVéhicules {
    Vector<Voiture> v;
    public static void main (String [] args){
        Random r = new Random();
        int n = 1+ r.nextInt(100);
        for(int i =0; i<n;i++)
            if(r.nextBoolean()){
            Voiture f = new Voiture(3*r.nextFloat(),
            3*r.nextFloat(), 3*r.nextFloat(),3*r.nextFloat());
            v.addElement(f);
        }
    System.out.println(v);
    }
}
```

Map : tableau associatif

- Type générique : Map<K, V>
  - K : type des clés
  - V : type des valeurs

```
V put(K k, V v)          // ajouter v avec la clé k (ou remplacer)
V get(Object k)          // la valeur associée à la clé ou null
V remove(Object k)       // supprimer l'entrée associée à k

boolean containsKey(Object k)   // k est-elle une clé utilisée ?
boolean containsValue(Object v) // v est-elle une valeur de la table ?

Set<Map.Entry<K, V>> entySet()  // toutes les entrées de la table
Set<K>               keySet()   // l'ensemble des clés
Collection<V>        values()   // la collection des valeurs

void putAll(Map<? extends K,? extends V> m)  // ajouter les entrées de m

int size()               // nombre d'entrées dans la table
boolean isEmpty()        // la table est-elle vide ?
void clear()             // vider la table
```

Exemple d'utilisation des Map

```
1  import java.util.*;
2  public class CompteNbOccurrences {
3      /** Compter le nombre d'occurrences des chaînes de args... */
4      public static void main(String[] args) {
5          Map<String, Integer> occ = new HashMap<String, Integer>();
6          for (String s : args) {
7              int ancien = occ.containsKey(s) ? occ.get(s) : 0;
8              occ.put(s, ancien + 1);
9          }
10         System.out.println("occ = " + occ);
11         System.out.println("clés = " + occ.keySet());
12         System.out.println("valeurs = " + occ.values());
13         System.out.println("entrées = " + occ.entrySet());
14
15         // afficher chaque entrée
16         for (Map.Entry<String, Integer> e : occ.entrySet()) {
17             System.out.println(e.getKey() + " --> " + e.getValue());
18         }
19     }
20 }
```

Exemple d'utilisation des Map (exemple)

Le résultat de `java CompteNbOccurrences A B C A C D E A A D E` est :

```
occ = {D=2, E=2, A=4, B=1, C=2}
clés = [D, E, A, B, C]
valeurs = [2, 2, 4, 1, 2]
entrées = [D=2, E=2, A=4, B=1, C=2]
D --> 2
E --> 2
A --> 4
B --> 1
C --> 2
```