Part III

Les collections

Types énumérés

Avant

```
public class Vegetable {
    public static final int carotte = 1;
    public static final int broccoli = 2;
    public static final int salade = 3;
}
public class Repas {
    public void servir (int v) {
        switch ( v ) {
            case Vegetable.broccoli:  /* ...  */ break;
            case Vegetable.carotte:  /* ...  */ break;
            ...
        }
    }
}
```

Après (depuis Java 1.5)

```
enum Vegetable {carotte,broccoli,salade} ;
for (Vegetable v :  Vegetable.values()) {
    System.out.println(v);
}
```

Les collections

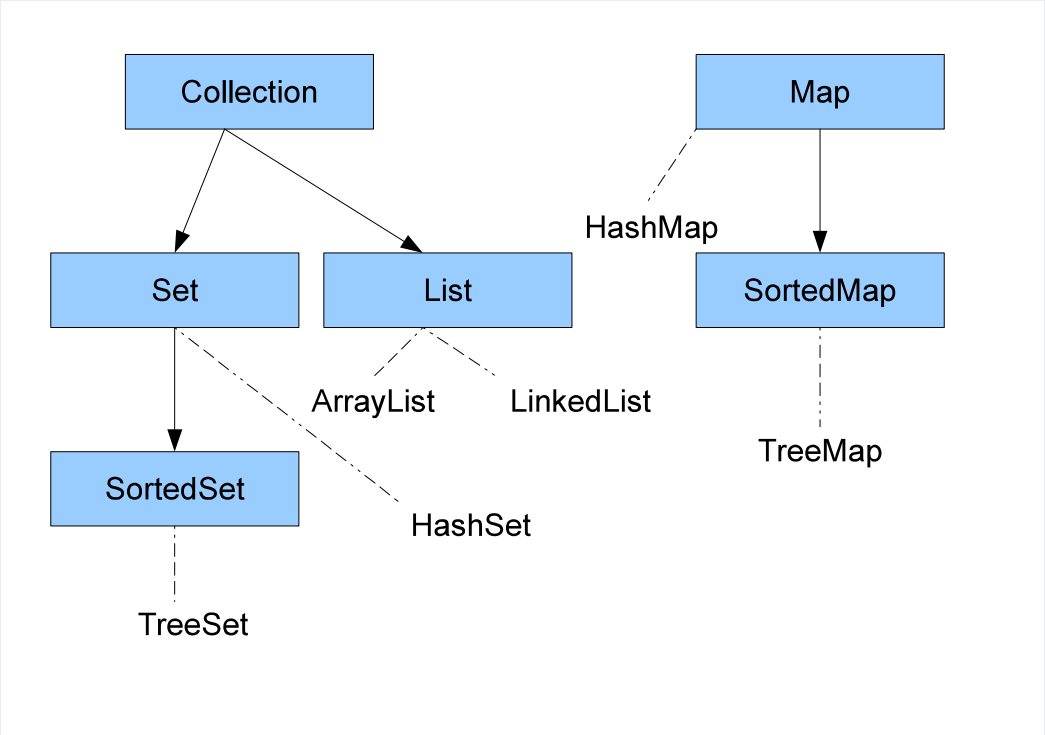
Definition (Collection)

Structure de données de taille variable :

- permettant de stocker et de manipuler différents éléments
- ne contenant que des objets, potentiellement de classes différentes
- Ex : main de poker, répertoire mail, répertoire téléphonique, ...

- Le Collection Framework (package `java.util`) regroupe :
  - des interfaces
  - des implémentations (classes concrètes)
- Il confère les avantages suivants au programmeur :
  - algorithmes prédéfinis
  - effort de programmation réduit
  - une meilleure qualité et une meilleure vitesse d'exécution des programmes
  - une standardisation (réutilisation des API et interopérabilité du code)

La hiérarchie des collections



- `Collection` : interface racine
- `Set` : collection non ordonnée sans duplicata
- `List` : collection ordonnée ($\neq$ `LinkedList`)
- `Map` : associe au plus une valeur à une clé donnée :
  - `Map` n'est pas une collection mais y est assimilé car il fournit les mêmes méthodes.

L'interface Collection

Definition

```
public interface Collection{
   //Opérations de base
   int size();
   boolean isEmpty();
   boolean contains(Object element);
   boolean add(Object element);
   boolean remove(Object element);
   Iterator iterator();
   //Opérations d'ensemble (Bulk operations)
   boolean containsAll(Collection c);
   boolean addAll(Collection c);
   boolean removeAll(Collection c);
   boolean retainAll(Collection c);
   void clear();
   //Opérations sur les Array
   Object[] toArray();
   Object[] toArray(Object a[]);
}
```

Exemples de bulk operations

- `c.retainAll(d);` :
  - élimine tous les éléments de `c` qui ne sont pas dans `d`
- `c.removeAll(Collections.singleton(e));` :
  - élimine toutes les instances de `e` dans `c`
- `c.removeAll(Collections.singleton(null));` :
  - élimine tous les éléments `null` dans `c`
- `Object[] a = c.toArray();` :
  - copie tous les éléments de `c` dans le tableau `a`
- `String[] b= new String[13]; String[] a = (String[]) c.toArray(b);` :
  - remplit le tableau `b` avec les éléments de la collection `c`
    - ★ la taille du tableau doit être égale au nombre d'éléments dans la collection `c`

La classe Collections

Definition

Classe fournissant des méthodes utilitaires (de manière non-exhaustive) :

- `public static Object min(Collection c)`
- `public static Object min(Collection c, Comparator comp)`
- `public static Object max(Collection c)`
- `public static Object max(Collection c, Comparator comp)`
- `public static void copy(List list1, List list2)`
- `public static void reverse(List list)`
- `public static void shuffle(List list)`
- `public static void sort(List list)`

L'interface Comparable

Definition (Comparable)

- `sort` permet de trier une liste `l` :
  - `Collections.sort(l);`
- Les éléments de la liste doivent implémenter l'interface `Comparable`
- `Comparable` fournit un ordre naturel sur les classes qui l'implémentent

```
public interface Comparable{
   public int compareTo(Object o);
}
```

L'ordre dépend du type de données :

- `String` : alphabétique
- `Date` : chronologique
- `Byte`, `Character`, `Long`, `Integer`, `Short`, `Double`, `Float` : ordre numérique
- `File` : ordre lexicographique sur les chemins de fichiers

L'interface Set

Definition (Set)

`Set` : interface collection interdisant la duplication d'éléments.

`Set` possède deux implémentations :

- `HashSet` :
  - qui stocke ses éléments dans une table de hachage (`Hashtable`).
- `TreeSet` (hérite de `SortedSet`) :
  - qui stocke ses éléments dans un arbre binaire bien équilibré.

Exemple d'utilisation de Set

Example (Impression des duplicats, du nombre et de la liste de mots distincts)

```
import java.util.*;
public class TrouverLesDuplicats{
   public static void main(String{[] args){
      Set s = new HashSet();
      for (int i=0; i<args.length;i++){
         if(!s.add(args[i])){
            System.out.println(“Duplicat”+args[i]);
         }
      }
      System.out.prinln(s.size()+ “ mots distincts :  ”+s);
   }
}
```

Remarque

Il est préférable d'utiliser `Set` au lieu de `HashSet` pour la référence (plus grande portabilité).

L'interface List

Definition (List)

L'interface List (parfois appelée séquence) est une collection ordonnée pouvant contenir des duplicats ("généralisation" de Vector).

```
public interface List extends Collection{
   //Accès
   Object get(int index);
   Object set(int index, Object element);
   void add(int index, Object element);
   Object remove(int index);
   boolean addAll(int index, Collection c);
   //Recherche
   int indexOf(Object o);
   int lastIndexOf(Object o);
   //Iteration
   ListIterator listIterator();
   ListIterator listIterator(int index);
   //Vue
   List subList(int from, int to);
}
```

L'interface List

Les méthodes héritées de l'interface `Collection` :

- `get, set, remove, add` :
  - équivalentes à `elementAt, setElementAt, removeElementAt, insertElementAt` dans `Vector`
- `add` et `addAll` :
  - ajoutent les éléments en fin de liste
  - exemple : concaténation `liste1.addAll(liste2);`
- `remove` :
  - supprime le premier élément

Example (Permutation de deux éléments dans une liste)

```
public static void swap(List a, int i, int j){
   Object tmp=a.get(i);
   a.set(i,a.get(j));
   a.set(j,tmp);
}
```

Utilisations de List

Example (Méthode shuffle de la classe Collections)

- Algorithme juste (fair : équiprobable) et rapide (taille de la liste -1)

```
import java.util.*;
public static void shuffle(List list, Random rnd){
   for (int i=liste.size();i>1;i--){
      swap(list,i-1,rnd.nextInt(i));
   }
}
```

Deux implémentations de l'interface `List` :

- `ArrayList` :
  - "représentation d'une liste en terme de `Vector`"
  - Avantage : accès à un élément en temps constant (contre linéaire pour `LinkedList`)
- `LinkedList` :
  - Liste chainée
  - Avantage : suppression (intérieure) d'un élément en temps constant (contre linéaire pour `ArrayList`)

L'interface Map

Definition (Map)

- Un `Map` est une application associant une clé à une valeur.
- Les clés ne peuvent pas être dupliquées : à chaque clé correspond une seule valeur.
- Il existe 3 implémentations de `Map` :
  - `HashMap`,
  - `TreeMap`
  - et aussi `Hashtable` (dans les dernières versions de Java)

L'interface Map

Definition (Map)

```
public interface Map {
    //Opérations de base
    Object get(Object key);
    Object put(Object key, Object value);
    Object remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();
    //Opérations d'ensemble
    void putAll(Map t);
    void clear();
    //Vues
    public Set KeySet();
    public Collection values();
    public Set entrySet();
    //Interface entrySet
    public interface Entry{
    Object getKey();
    Object getValue();
    Object setValue(Object value);
    }
}
```

Comparaison entre Hashtable et Map

Différences entre Map et Hashtable :

- Map fournit des vues au type Collection en lieu et place d'Enumeration
- Map permet d'itérer sur les paires (contrairement à Hashtable)
- Map permet de supprimer des entrées de manière sûre lors d'une itération

Similitudes :

- put, get, remove, containsKey, containsValue, size et isEmpty se comportent comme dans Hashtable

Utilisation de Map

Example (Programme associant chaque mot au nombre de fois où il apparait en argument de la ligne de commande)

```
import java.util.*;
public class Frequence{
   private static final Integer ONE = new Integer(1);
   public static void main(String[] args){
      Map m = new HashMap();
      for (int i=0;i<args.length;i++){
         Integer freq = (Integer) m.get(args[i]);
         m.put(args[i], (freq==null ?  ONE : new
Integer(freq.intValue()+1)));
      }
      System.out.println(m.size()+"mots distincts :  ");
      System.out.println(m);
   }
}
```

L'interface Map

- Les opérations d'ensemble (bulk operations) :
  - `putAll` et `clear` se comportent de manière classique
    - ★ comme `addAll` et `clear` pour les collections
- Les vues :
  - `KeySet` : retourne un `Set` de clés
  - `values` : retourne une `Collection` de valeurs
    - ★ Ce n'est pas un `Set` car plusieurs clés peuvent être associées à la même valeur.
  - `entrySet` : retourne un `Set` de couples clé-paire de type `Entry`

L'interface SortedSet (et SortedMap)

Definition (SortedSet)

Un SortedSet est un Set dont les éléments sont triés :

- dans l'ordre naturel croissant
- ou suivant un ordre spécifié par un Comparator.

```
public interface SortedSet extends Set{
   //Vues
   SortedSet subSet(Object fromElement, Object toElement);
   SortedSet headSet(Object toElement);
   SortedSet tailSet(Object fromElement);
   //Début et fin
   Object first();
   Object last();
   //Accès au comparateur
   Comparator comparator();
}
```

Interface Comparator

Definition (Comparator)

L'interface `Comparator` est définie par :

```
public interface Comparator{
   int compare(Object o1, Object o2);
}
```

Retourne :

- `-1` si `o1` est plus petit que `o2`
- `0` si `o1` et `o2` sont égaux
- `1` si `o1` est plus grand que `o2`

Exemple d'utilisation de Comparator

Example (Classe Anonyme avec Comparator)

```
import java.util.*;
public class Anonyme {
    public static void main(String args[]) {
        List list = new ArrayList();
        list.add(new Integer(37));
        list.add(new Integer(-59));
        list.add(new Integer(83));
        Collections.sort(list);
        System.out.println(list); // -59 37 83
        Collections.sort(list, new Comparator() {
            public int compare(Object o1, Object o2) {
                int a = ((Integer)o1).intValue();
                int b = ((Integer)o2).intValue();
                if (a < b) return 1;
                else if (a == b) return 0;
                else return -1;
            }
        });
        System.out.println(list); // 83 37 -59
    }
}
```

L'interface SortedSet (et SortedMap)

Definition (SortedSet)

Les implémentations de `SortedSet` possèdent trois constructeurs :

- `public Implementation (Collection c)`
- `public Implementation (Comparator comp)`
- `public Implementation (SortedSet ss, Comparator comp)`

`TreeSet` est une implémentation de `SortedSet` qui diffère de `HashSet` :

- moins efficace pour les opérations générales (temps logarithmique contre temps constant)
- mais plus efficace pour des opérations ordonnées

Remarque

L'interface `SortedMap` se comporte de manière identique.

Résumé de la situation

Implémentations					
Interfaces		Hash Table	Tableau variable	Arbre bien balancé	Liste chaînée
	Set	HashSet		TreeSet	
	List		ArrayList		LinkedList
	Map	HashMap		TreeMap	

L'interface Iterator

Definition (Iterator)

L'interface `Collection` déclare une méthode `iterator` retournant un objet d'une classe implémentant l'interface `Iterator` :

```
public interface Iterator{
   boolean hasNext();
   Object next();
   void remove();
}
```

Example

```
static void filtre(Collection c){
   for(Iterator i=c.iterator(); i.hasNext()){
      if(!cond(i.next())){
         i.remove();
      }
   }
}
```

Exemple d'Iterator (suite)

Example

```
Iterator iter = c.iterator();
while (iter.hasNext()) {
   Objet o = iter.next();
   System.out.println(o);
}
```

Itération sur List et Map

Definition (List possède un itérateur plus puissant)

```
public interface ListIterator extends Iterator{//...
   boolean hasPrevious();
   Object previous();
   int nextIndex();
   int previousIndex();
   //...
}
```

Example (Parcours d'une liste en sens inverse)

```
for (ListIterator i=l.listIterator(l.size());l.hasPrevious();){
   ...= l.previous();
}
```

Example (Les vues sont l'unique moyen d'itérer sur les Map)

```
for (Iterator i=m.keySet().iterator(); i.hasNext();){
   System.out.println(i.next());
}
```

Autres structures désuètes de `java.util`

- Les Enumerations : classe `Enumeration`
  - équivalent de `Iterator`
- Les vecteurs : classe `Vector`
  - analogue de `ArrayList`
- Les piles : classe `Stack`
  - vecteurs spéciaux incluant des méthodes `push` et `pop` pour ajouter et supprimer un élément
- Les dictionnaires : classe `Dictionary`
  - analogue de `Map` qui n'est pas une interface mais une classe abstraite
- Les tables de hachage : classe `Hashtable`
  - analogue de `HashMap`
- Les propriétés système : classe `Properties`
  - `Hashtable` où clés et valeurs sont de type `String` définissant les propriétés système
  - Encore utilisée

Généricité

Definition (Type générique)

On peut généraliser une classe à différents types en utilisant les types génériques :

```
class ArrayList<E>{
   ...
}
```

Example

On spécifie le type passé en paramètre lors de l'utilisation de la classe :

```
ArrayList<Integer> list = new ArrayList<Integer>();
```

Remarque

Les collections sont toutes génériques depuis Java 1.5.

Généricité

Example (Exemple d'erreur à la compilation)

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(0,new Integer(25));
list.add(0,new Double(2.5));//erreur
```

Example (Définition d'un type générique)

```
public class Gen<X,Y>{
   public X premier;
   public Y second;
   public Gen(X a, Y b){
      premier=a;
      second=b;
   }
   public X getPremier(){
      return premier;
   }
   ...
}
```

La WildCard ?

? représente n'importe quel type

Example

```
Gen<?,?> a;
Gen<String,Double> a = new Gen<String,Double>(“bonjour”, new
Double(3.3));
Gen<Integer,String> a = new Gen<Integer,String>(new
Integer(666),“au revoir”);
```

- Collections hétérogènes :
  - `Collection<Object>` :
    - ★ peut contenir différents types
- Collections homogènes :
  - `Collection<?>` :
    - ★ contient un seul type non spécifié

Encore quelques Remarques

Example (Classe instanciable par toute classe implémentant l'interface Comparable)

```
public class Classe<X implements Comparable<X>>{
   ...
   public static void main(String[] args){
      Classe<String> e = new Classe<String>();
      ...
   }
}
```

Example (Types génériques non instanciables avec des types primitifs)

```
public class Classe<X>{
   ...
   public static void main(String[] args){
      Classe<int[]> e = new Classe<int[]>();//ok
      Classe<int> e= new Class<int>();//erreur
      ...
   }
}
```

foreach

`for (type variable : objet) {instructions}`

- `instructions` est exécutée pour chaque composant d'`objet`
- Ne fonctionne que pour les objets composés :
  - tableaux, énumérations, collections...

Example (foreach)

```
ArrayList a = new ArrayList();
a.add(25); a.add(39); a.add(54);
x=0;
for(int i :  a){
   System.out.println(“Prix de la bière ”+x+” est “+i+” euros “);
   x++;
}
```

A préférer à la boucle `for` car évite les erreurs d'indice très fréquentes dans les programmes.