Programmation orientée objet

Chapitre9: Les collections

Définition

- Les principales structures de données des classes utilitaires **(java.util)** sont regroupées sous une même interface **Collection**. Cette interface est implémenté par des **ensembles**, **vecteurs dynamiques** et **tables associatives**.
- Une collection fournit un ensemble de méthodes qui permettent:
  - ✓ D'ajouter un nouveau objet dans le tableau
  - ✓ Supprimer un objet du tableau
  - ✓ Rechercher des objets selon des critères
  - ✓ Trier le tableau d'objets
  - ✓ Contrôler les objets du tableau
  - ✓ Etc…
- Dans un **problème**, les tableaux peuvent être utilisés quand la dimension du tableau est **fixe**.
- Dans le cas contraire, il faut utiliser les collections.

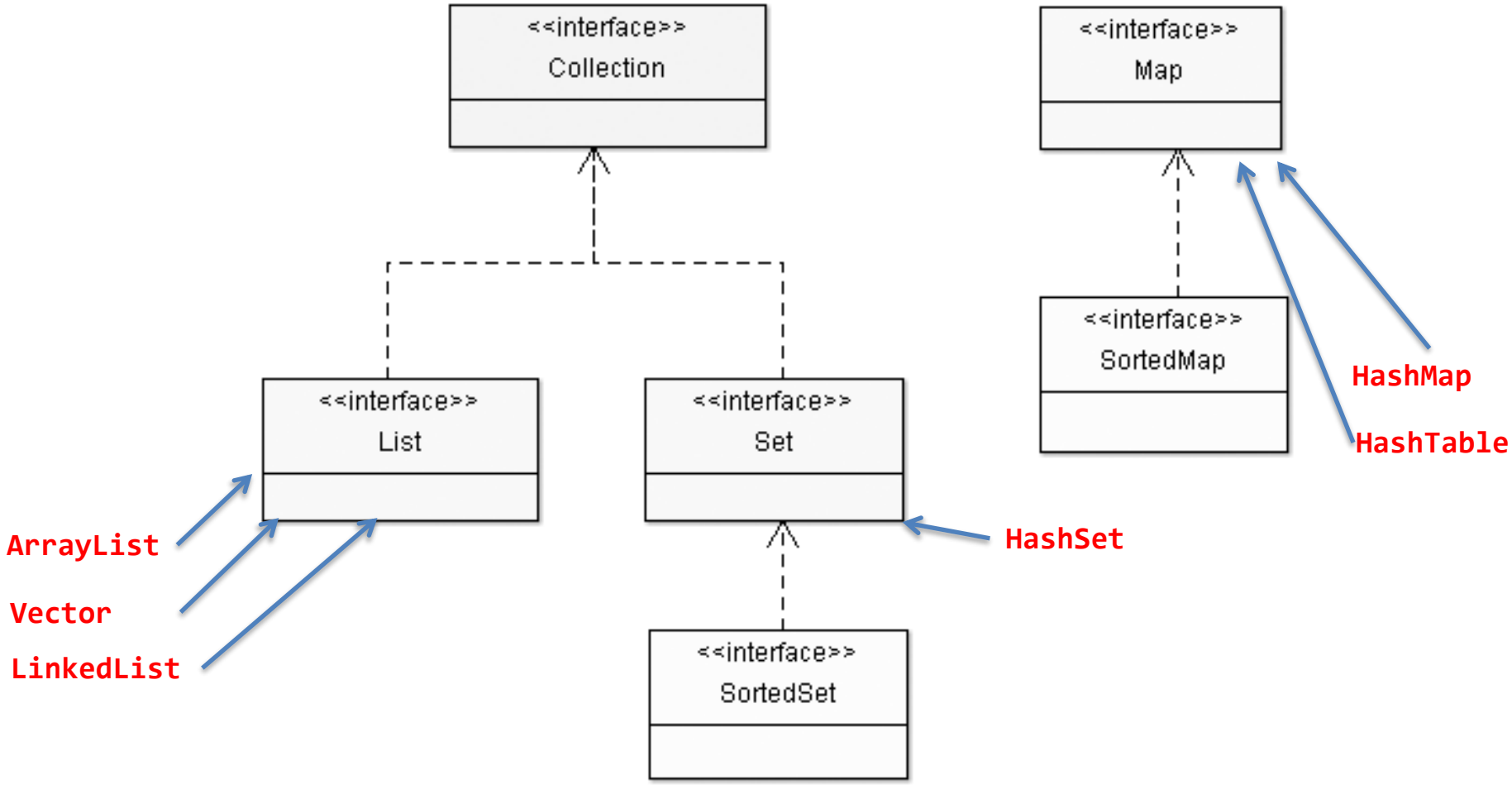
Définition

- Java fournit plusieurs types de collections:
  - ✓ ArrayList
  - ✓ Vector
  - ✓ Iterator
  - ✓ HashMap
  - ✓ Etc…

L'API de collections propose un ensemble d'interfaces dont le but est de stocker de multiples objets. Elle propose 3 types de collections.

- ✓ **List:** collection d'éléments ordonnés qui accepte les doublons.
- ✓ **Set:** collection d'éléments non ordonnés par défaut qui n'accepte pas les doublons.
- ✓ **Map:** collection sous la forme d'une association de paires clé/valeur

Interfaces utilisées dans les collections



Quelques méthodes d'une collection de type List

Quelques méthodes disponibles dans l'interface Collection:

- ✓ **int size()** : retourne le nombre d'éléments portés par cette collection
- ✓ **boolean isEmpty()**: un booléen qui permet de tester si cette collection est vide ou pas.
- ✓ **boolean contains(T t)** : retourne true si l'objet passé en paramètre est contenu dans cette collection.
- ✓ **boolean add(T t)** et **boolean remove(T t)** : permet d'ajouter (resp. de retirer) un objet à cette collection.
- ✓ **void clear()** : efface la collection courante.
- ✓ **Object[] toArray()** : convertit la collection courante en tableau d'objets.

Voyons les méthodes que l'interface List ajoute à Collection:

- ✓ **void add(int index, T t)** : permettent d'insérer un élément à la position notée par index.
- ✓ **T set(int index, T t)** : permet de remplacer l'élément placé à la position index par celui passé en paramètre. L'élément qui existait est **retiré** de la liste, et **retourné** par cette méthode.
- ✓ **T get(int index)** : retourne l'élément placé à l'index passé en paramètre.
- ✓ **T remove(int index**) : retire l'élément placé à l'index passé en paramètre. Cet élément est **retourné** par la méthode.
- ✓ **int indexOf(Object o)** : retournent respectivement le premier index de l'objet passé en paramètre dans cette liste.

Les collections de type List

ArrayList:

ArrayList est une classe du package java.util, qui implémente l'interface **List**.

- Déclaration d'une collection de type List qui devrait stocker des objets de type Fruit:

```
List<Fruit> fruits;
```

- Création de la liste:

```
fruits=new ArrayList<Fruit>();
```

Ajouter deux objets de type Fruit à la liste:

```
fruits.add(new Pomme(30));
fruits.add(new Orange(25));
```

- Faire appel à la méthode affiche() de tous les objets de la liste:

✓ En utilisant la boucle classique for

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

✓ En utilisant la boucle **for each**

```
for(Fruit f:fruits)
        f.affiche();
```

- Supprimer le deuxième Objet de la liste

```
fruits.remove(1);
```

Exemple d'utilisation de ArrayList

```
import java.util.ArrayList;
import java.util.List;
public class App1 {
 public static void main(String[] args) {
  // Déclaration d'une liste de type
Fruit
  List<Fruit> fruits;
  // Création de la liste
  fruits=new ArrayList<Fruit>();
  // Ajout de 3 objets Pomme, Orange et
Pomme à la liste
  fruits.add(new Pomme(30));
  fruits.add(new Orange(25));
  fruits.add(new Pomme(60));
```

```
  // Parcourir tous les objets
  for(int i=0;i<fruits.size();i++){
  // Faire appel à la méthode affiche() de
chaque Fruit de la
liste
  fruits.get(i).affiche();
  }
  // Une autre manière plus simple pour
parcourir une liste
  for(Fruit f:fruits) // Pour chaque Fruit de
la liste
  f.affiche(); // Faire appel à la méthode
affiche() du Fruit f
 }
}
```

Vector

Vector est une classe du package java.util qui fonctionne comme **ArrayList**

- Déclaration d'un Vecteur qui devrait stocker des objets de type Fruit:

```
Vector<Fruit> fruits;
```

- Création de la liste:

```
fruits=new Vector<Fruit>();
```

- Ajouter deux objets de type Fruit à la liste:

```
fruits.add(new Pomme(30));
fruits.add(new Orange(25));
```

- Faire appel à la méthode affiche() de tous les objets de la liste:

✓ En utilisant la boucle classique for

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

✓ En utilisant la boucle **for each**

```
for(Fruit f:fruits)
        f.affiche();
```

- Supprimer le deuxième Objet de la liste

```
fruits.remove(1);
```

Exemple d'utilisation de Vector

```
import java.util.Vector;
public class App2 {
 public static void main(String[] args) {
  // Déclaration d'un vecteur de type Fruit
  Vector<Fruit> fruits;
  // Création du vecteur
  fruits=new Vector<Fruit>();
  // Ajout de 3 objets Pomme, Orange et Pomme au vecteur
  fruits.add(new Pomme(30));
  fruits.add(new Orange(25));
  fruits.add(new Pomme(60));
```

```
  // Parcourir tous les objets
  for(int i=0;i<fruits.size();i++){
  // Faire appel à la méthode affiche() de chaque Fruit
   fruits.get(i).affiche();
  }
  // Une autre manière plus simple pour parcourir un vecteur
  for(Fruit f:fruits) // Pour chaque Fruit du vecteur
   f.affiche(); // Faire appel à la méthode affiche() du Fruit f
 }
}
```

Collection de type Iterator

- La collection de type **Iterator** du package java.util est souvent utilisée pour afficher les objets d'une autre collection.
- En effet il est possible d'obtenir un iterator à partir de chaque collection.

Exemple :

- Création d'un vecteur de Fruit.

```
Vector<Fruit> fruits=new Vector<Fruit>();
```

- Ajouter des fruits aux vecteur

```
fruits.add(new Pomme(30));
fruits.add(new Orange(25));
fruits.add(new Pomme(60));
```

- Création d'un Iterator à partir de ce vecteur

```
Iterator<Fruit> it=fruits.iterator();
```

- Parcourir l'Iterator:

```
while(it.hasNext()){
  Fruit f=it.next();
  f.affiche();}
```

- Notez bien que, après avoir parcouru un iterator, il devient **vide**

Collection de type ListIterator

- La collection de type **ListIterator** du package java.util est souvent utilisée pour afficher les objets d'une autre collection de type **List**.

Exemple :

- Création d'un vecteur de Fruit.

```
Vector<Fruit> fruits=new Vector<Fruit>();
```

- Ajouter des fruits aux vecteur

```
fruits.add(new Pomme(30));
fruits.add(new Orange(25));
fruits.add(new Pomme(60));
```

- Création d'un Iterator à partir de ce vecteur

```
ListIterator<Fruit> it=fruits.listIterator();
```

- Parcourir l'Iterator:

```
while(it.hasNext()){
  Fruit f=it.next();
  f.affiche();
}
```

Les ensembles (HashSet)

HashSet:

La classe **HashSet** implémente la notion d'ensemble. Un ensemble est une collection **non ordonnée** d'éléments, un élément ne pouvant apparaître **qu'au plus une fois**.

Construction et parcours :

```
HashSet e1 = new HashSet(); // ensemble vide
Hashset e2 = new HashSet(c); // ensemble contenant tous
                             //les éléments d'une collection c
```

Un parcours d'un ensemble se fait à l'aide d'un itérateur :

```
HashSet e;
...
Iterator it = e.iterator();
while(it.hasNext()) {
 Object o = it.next();
 ....
}
```

HashSet

Ajout d'un élément :

```
HashSet e;
Object elem;
...
boolean existe = e.add(elem);
if(existe) Sytem.out.println(elem+" a été ajouté ");
else Sytem.out.println(elem+" existe déjà");
```

Suppression:

```
HashSet e;
Object elem;
...
boolean existe = e.remove(elem);
if(existe) Sytem.out.println(elem+" a été supprimé");
else Sytem.out.println(elem+" n'existe pas");
```

Autre possibilité de suppression :

```
HashSet e;
...
Iterator it = e.iterator();
it.next(); it.next();it.remove();
```

HashSet

Opérations ensemblistes :

– e1.addAll(e2) place dans e1 tous les éléments de e2.

– e1.retainAll(e2) garde dans e1 tout ce qui appartient à e2.

– e1.removeAll(e2) supprime de e1 tout ce qui appartient à e2.

Notion de Hachage

Un ensemble **HashSet** est implémenté par une table de hachage, c'est-à-dire que ses éléments sont stockés selon une position donnée. Cette position est définie selon un code calculé par la méthode int hashCode() utilisant la valeur effective des objets.

Les classes String et File par exemple implémentent déjà cette méthode. Par contre les autres classes utilisent par défaut une méthode dérivée de Object qui se content d'utiliser comme valeur la simple **adresse** des objets (dans ces conditions 2 objets de même valeur auront toujours des codes de hachage différents). Si l'on souhaite définir un ordre des éléments basés sur leur valeur effective, il faut **redéfinir** la méthode hashCode dans la classe correspondante.

Exemple:

```
class Point {
private int x,y;
public int hashCode(){ return x+y;}
}
```

Tri des collection de type list

1ère méthode: L'interface Comparable

Tous les objets qui doivent définir un **ordre naturel** utilisé par le tri d'une collection **doivent** implémenter cette interface.

Cette interface ne définit qu'une **seule** méthode:

int compareTo(Object)qui doit renvoyer:

- une valeur <0 si l'objet courant est inférieur à l'objet fourni
- une valeur >0 si l'objet courant est supérieur à l'objet fourni
- une valeur = 0 si l'objet courant est égal à l'objet fourni

➔A l'aide de cette méthode de comparaison, nous pouvons trier une liste d'objets grâce à la méthode **Collections.sort(List l)**

➔On peut aussi rechercher le maximum et le minimum a l'aide de des méthodes **max**(List l) et **min**(List l)

➔La méthode Collections.shuffle(l); //mélange des éléments de la liste l

➔Collections.sort(l,Collections.reverseOrder()); //tri des éléments de l en ordre inverse

Remarque: String et Date implémentent déjà l'interface Comparable.

Tri des collection de type list

2ème méthode: L'interface Comparator

Cette interface représente un ordre de tri quelconque. Elle est utile pour permettre le tri d'objets qui n'implémentent pas l'interface Comparable ou pour définir un **ordre de tri différent** de ce lui défini avec Comparable.

Cette interface ne définit qu'une seule méthode: **int compare(Object, Object).** Qui compare les deux objets fournis en paramètre et renvoie:

- une valeur < 0 si le premier objet est inférieur au second
- une valeur > 0 si le premier objet est supérieur au second
- une valeur = 0 si les deux objets sont égaux

A l'aide de cette méthode de comparaison, nous pouvons trier une liste d'objets selon l'ordre précisé par l'objet Comparator grâce à la méthode **Collections.sort(List l, Comparator o)**

Les tables associatives (HashMap et HashTable)

Une table associative (ou de hachage) permet de conserver une information association deux parties nommées **clé** et **valeur**. Elle est principalement destinée à retrouver la valeur associée à une clé donnée. Les exemples les plus caractéristiques de telles tables sont :

- ✓ Le dictionnaire : à un mot (clé) on associe une valeur qui est sa définition,
- ✓ L'annuaire usuel : à un nom (clé) on associe une valeur comportant le numéro de téléphone et éventuellement une adresse,
- ✓ L'annuaire inversé : à un numéro de téléphone (qui devient la clé) on associe une valeur comportant le nom et éventuellement une adresse.

La collection HashMap

- La collection **HashMap** est une classe qui implémente l'interface **Map**. Cette collection permet de créer un tableau dynamique d'objet de type Object qui sont identifiés par une clé.
- Déclaration et création d'une collection de type HashMap qui contient des fruits identifiés par une clé de type String :

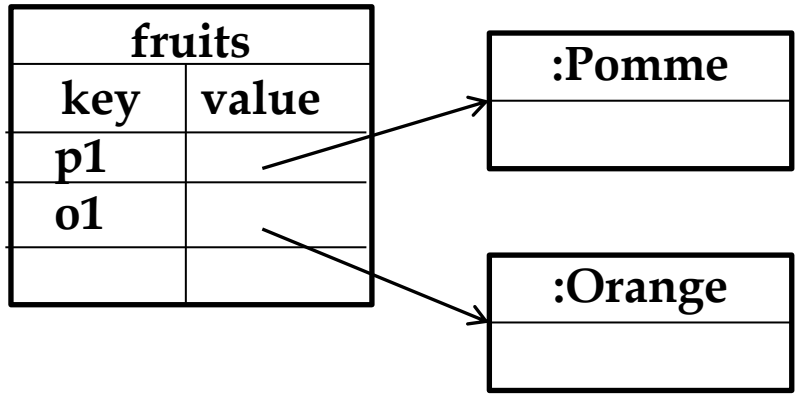
```
Map<String, Fruit> fruits=new HashMap<String, Fruit>();
```

- Ajouter deux objets de type Fruit à la colelction

```
fruits.put("p1", new Pomme(40));
fruits.put("o1", new Orange(60));
```

- Récupérer un objet ayant pour clé "p1"

```
Fruit f=fruits.get("p1");
f.affiche();
```



La collection HashMap

➢ Parcourir toute la collection:

```
Iterator<String> it=fruits.keySet().iterator();
while(it.hasNext()){
        String key=it.next();
        Fruit ff=fruits.get(key);
        System.out.println(key);
        ff.affiche();
}
```

La collection HashMap

- Parcourir toute la collection: notion de vue

```
HashMap m;
Set entrees = m.entrySet(); //entrees est un ensemble de paires
Iterator iter = entrees.iterator();
while(iter.hasNext()) {
  Map.Entry entree = (Map.Entry) iter.next(); //paire courante
  Object cle = entree.getKey(); //clé de la paire courante
  Object valeur = entree.getValue(); //valeur de la paire courante
... }
```

- L'ensemble renvoyé par **entrySet** n'est pas une copie de la table, c'est **une vue**.
- Autre vues :
  - ✓L'ensemble des clés :

```
HashMap m;
Set cles = m.keySet();
```

  - ✓La collection des valeurs :

```
Collection valeurs = m.values();
```

Classes et méthodes génériques

Si on souhaite définir une collection (Vector, ArrayList, ...) contenant un ensemble d'objets de sorte que tous ces objets soient de type **TypeElement**, on peut définir des collections ne contenant que ce type d'objet en déclarant : **List<TypeElement>**. Ceci permet de vérifier que l'ajout des éléments à la liste est bien de type TypeElement et que lors d'accès à des éléments de la liste on a la certitude qu'ils sont bien de type TypeElement.

Exemple:

```
Vector<Point> v=new Vector<Point>(); //déclaration
//on peut ajouter uniquement des objets de Point
//ici mon objet doit être obligatoirement de ce type
v.add(monobjet);
...
//on peut récupérer directement des objets Point
for(int i=0;i<v.size();i++){
 Point e=v.get(i);
 ...
}
```

Classes et méthodes génériques

➢ La réalisation de classes ou d'interfaces paramétrées utilise une syntaxe similaire :

```
public class Exemple<T>{
private T membre ;
public Exemple<T>(T m){
membre=m ;}
public void setMembre(T m) { //méthode générique
membre=m ;}
public void affiche(){
System.out.println(membre);}
}
```

Le ou **les** types paramétrés définis entre **<** et **>** dans le nom de la classe peuvent être utilisé dans le corps comme s'il s'agissait d'un type de donnée existant.

Une instance Exemple<String> permettra donc d'invoquer la méthode setMembre() avec un paramètre de type String.

La définition de plus types paramétrés est possible en les séparant par des virgules comme dans <K, V>.