Collections Java

Définition d'interface (rappel)

Une interface définit les méthodes que certaines classes doivent obligatoirement implémenter.

Une interface :

- ne contient aucune donnée;
- contient des signatures de méthodes;
- n'associe aucune implémentation à ses méthodes

Si une classe **implémente** une interface, elle doit contenir les méthodes décrites dans l'interface en y associant un traitement.

- une classe peut implémenter plusieurs interfaces;
- une interface ne peut pas être instanciée;
- une interface peut hériter d'une autre interface.

Définition d 'une collection

Une ***collection*** regroupe plusieurs données de même nature

- Exemples : promotion d'étudiants, sac de jouets, ...

Une ***structure collective*** implante une collection

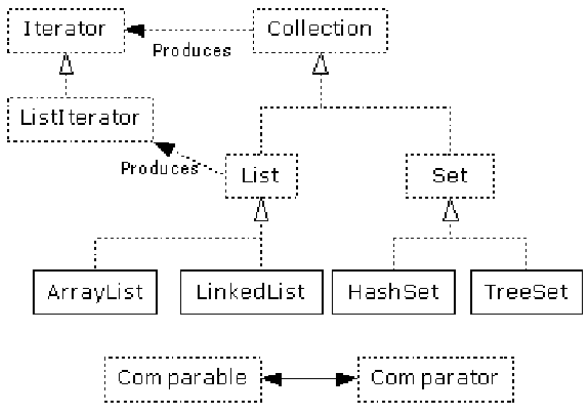
- plusieurs implantations possibles
  - ordonnées ou non, avec ou sans doublons, ...
  - accès, recherche, tris (algorithmes) plus ou moins efficaces

Objectifs

- adapter la structure collective aux besoins de la collection
- ne pas re-programmer les traitements répétitifs classiques (affichage, saisie, recherche d'éléments, …)

Vue d'ensemble des collections

Hiérarchie simplifiée



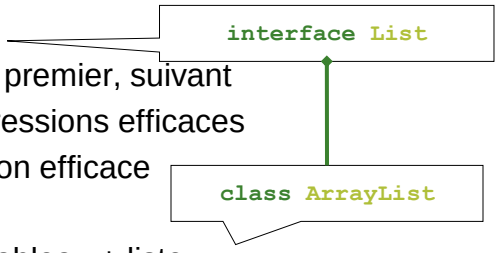
Structures collectives classiques

Tableau

`type[]` et `Array`

- accès par index
- recherche efficace si le tableau est trié (dichotomie)
- insertions et suppressions peu efficaces
- défaut majeur : nombre d'éléments borné

Liste



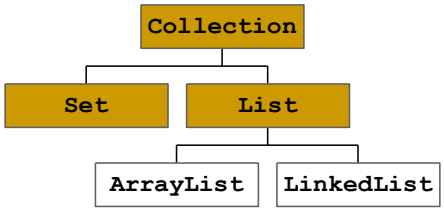
- accès séquentiel : premier, suivant
- insertions et suppressions efficaces
- recherche lente, non efficace

Tableau dynamique = tableau + liste

Paquetage `java.util`

Interface `Collection`

Interfaces `Set` et `List`



Méthodes

- `boolean add(Object o)`
- `boolean remove(Object o)`
- …

Plusieurs implantations

- tableau : `ArrayList`
- liste chaînée : `LinkedList`

- Algorithmes génériques : tri, maximum, copie ...
  - ♦ méthodes statiques de `Collection`

Collection : méthodes communes

`boolean add(Object)` : ajouter un élément
`boolean addAll(Collection)` : ajouter plusieurs éléments
`void clear()` : tout supprimer
`boolean contains(Object)` : test d'appartenance
`boolean containsAll(Collection)` : appartenance collective
`boolean isEmpty()` : test de l'absence d'éléments
`Iterator iterator()` : pour le parcours
`boolean remove(Object)` : retrait d'un élément
`boolean removeAll(Collection)` : retrait de plusieurs éléments
`boolean retainAll(Collection)` : intersection
`int size()` : nombre d'éléments
`Object[] toArray()` : transformation en tableau
`Object[] toArray(Object[] a)` : tableau de même type que a

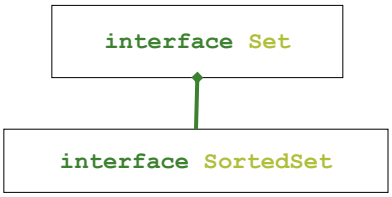
Exemple : ajout d'éléments

```
import java.util.*;

public class MaCollection {
   static final int N = 25000;
   List listEntier = new ArrayList();

   public static void main(String args[]) {
      MaCollection c = new MaCollection();
      int i;

      for (i = 0; i < N; i++) {
         c.listEntier.add(new Integer(i));
      }
   }
}
```

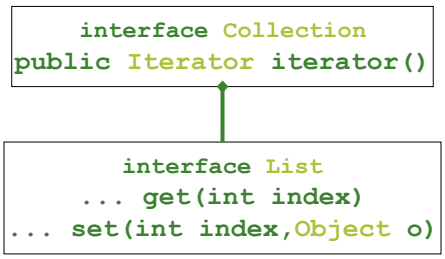
Caractéristiques des collections

Ordonnées ou non
- Ordre sur les éléments ? voir tri

Doublons autorisés ou non
- ***liste*** (`List`) : avec doubles
- ***ensemble*** (`Set`) : sans doubles



Besoins d'accès
- indexé
- séquentiel, via `Iterator`



Fonctionnalités des Listes

Implantent l'interface `List`

- `ArrayList`
  - Liste implantée dans un tableau
  - accès immédiat à chaque élément
  - ajout et suppression lourdes
- `LinkedList`
  - accès aux éléments lourd
  - ajout et suppression très efficaces
  - permettent d'implanter les structures FIFO (file) et LIFO (pile)
  - méthodes supplémentaires : `addFirst()`, `addLast()`, `getFirst()`, `getLast()`, `removeFirst()`, `removeLast()`

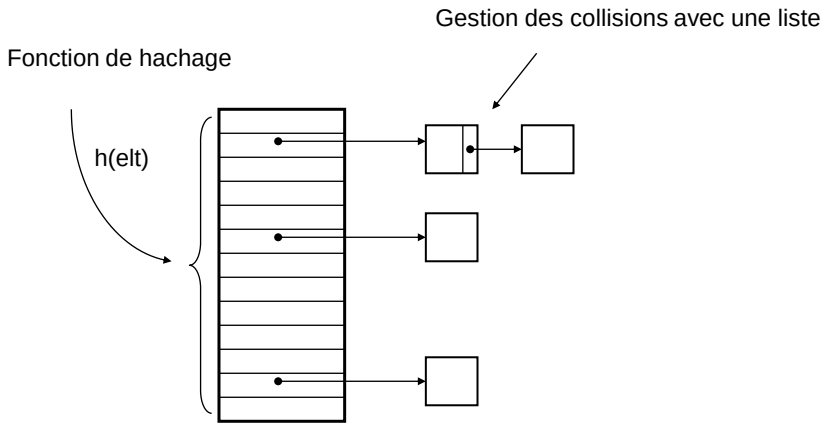
Fonctionnalités des ensembles

Implantent l'interface **`Set`**

Eléments non dupliqués

- **`HashSet`**
  - table de hashage
  - utiliser la méthode **`hashCode()`**
  - accès très performant aux éléments
- **`TreeSet`**
  - arbre binaire de recherche
  - maintient l'ensemble trié en permanence
  - méthodes supplémentaires
    - first() (mini), last() (maxi), subSet(deb,fin), headSet(fin), tailSet(deb)

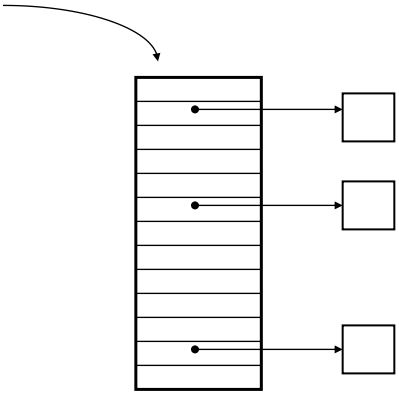
Une table de hachage, en général



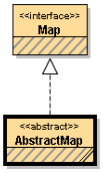
Une table instance de Map

Map table = new HashMap()
table.put(clé,valeur);

valeur = table.get(clé);



Adressage associatif, Hashtable



Map

+clear() : void
+containsKey(key : Object) : boolean
+containsValue(value : Object) : boolean
+entrySet() : Set
+get(key : Object) : Object
+isEmpty() : boolean
+keySet() : Set
+put(key : Object, value : Object) : Object
+putAll(mapping : Map) : void
+remove(key : Object) : Object
+size() : int
+values() : Collection

Map.Entry

+equals(object : Object) : boolean
+getKey() : Object
+getValue() : Object
+hashCode() : int
+setValue(value : Object) : Object

Recherche d'un élément

Méthode

- `public boolean contains(Object o)`
- interface `Collection`, redéfinie selon les sous-classes

Utilise l'égalité entre objets

- égalité définie par `boolean equals(Object o)`
- par défaut (classe `Object`) : égalité de références
- à redéfinir dans chaque classe d'éléments

Cas spéciaux

- doublons : recherche du premier ou de toutes les occurrences?
- structures ordonnées : plus efficace, si les éléments sont comparables (voir tri)

Tri d'une structure collective

Algorithmes génériques

- Collections.sort(List l)
- Arrays.sort(Object[] a,…)

Condition : collection d'éléments dont la classe définit des règles de comparaison

- en implémentant l'interface `java.lang.Comparable`
  - implements Comparable
- en définissant la méthode de comparaison
  - public int compareTo(Object o)
  - `a.compareTo(b) == 0` si `a.equals(b)`
  - `a.compareTo(b) < 0` pour `a` strictement inférieur à `b`
  - `a.compareTo(b) > 0` pour `a` strictement supérieur à `b`

Stack :

La classe Stack (dérivée de Vector) représente une pile d'objets que l'on gère en LIFO (last-in-first-out : dernier entré , premier sorti).

Méthodes :

Object push(Object item);	empile l'objet item (retourne l'objet empilé)
Object pop();	dépile l'objet au sommet
Object peek();	retourne l'objet au sommet (sans le dépiler)
Boolean empty();	retourne true si la pile est vide.

Exemple

```
import java.util.Stack;
class Pile {

static int sommeEntiers = 0;

 static Stack pile = new Stack();
```

```
static void placerDansPile(String[] listeArguments)
 {
   for (int i = 0; i < listeArguments.length; i++)
    {
                try
                 {
                   pile.push(Integer.valueOf(listeArguments[i]));
                 }
                catch(NumberFormatException e1)
                 {
                   try
                     {
                                pile.push(Double.valueOf(listeArguments[i]));
                     }
                   catch(NumberFormatException e2)
                     {
                       System.out.println("L'argument " + listeArguments[i] +
                                          " n'est ni un double ni un int");
                     }
                 }
    }
 }
```

Exemple (suite)

```
static void SommeEntiers()
 {
   while(!pile.empty())
    {
              Object objet;

              objet = pile.pop();
              System.out.println("On retrouve
une instance de “+
objet.getClass()+" qui vaut "+objet);
              if (objet instanceof Integer)
                 sommeEntiers +=
((Integer)objet).intValue();
    }
 }
```

```
public static void main(String[] argv)
 {
   placerDansPile(argv);
   System.out.println();
   SommerEntiers();
   System.out.println();
   System.out.println("La somme de vos entiers est "+
sommeEntiers);
 }
}
```

Pour la commande :

java Pile 5 6.4 coucou 3.0 2
on obtient :

```
L'argument coucou n'est ni un double ni un int

On retrouve une instance de class java.lang.Integer qui vaut 2
On retrouve une instance de class java.lang.Double qui vaut 3.0
On retrouve une instance de class java.lang.Double qui vaut 6.4
On retrouve une instance de class java.lang.Integer qui vaut 5

La somme de vos entiers est 7
```

Détails sur les listes

java.util.ArrayList :

Un ArrayList est un tableau qui se redimensionne automatiquement. Il accepte tout type d'objets, null y compris. Chaque instance d'ArrayList a une capacité, qui définit le nombre d'éléments qu'on peut y stocker. Au fur et à mesure qu'on ajoute des éléments et qu'on dépasse la capacité, la taille augmente en conséquence.

ArrayList n'est pas Thread Safe.

Un ArrayList fournit un accès aux éléments par leur indice très performant et est optimisé pour des opérations d'ajout/suppression d'éléments en fin de liste.

Complexité : Les opérations size, isEmpty, get, set, iterator sont exécutées en temps constant.

Les opérations d'ajout/suppression sont exécutées en temps constant amorti (les ajouts/suppressions en fin de liste sont plus rapides).

java.util.LinkedList :

Un java.util.LinkedList utilise une liste chainée pour ranger les données.

L'ajout et la suppression d'éléments est aussi rapide quelle que soit la position, mais l'accès aux valeurs par leur indice est très lente.

Complexité : Les opérations size, isEmpty, add, remove, set, get sont exécutées en temps constant. Toutes les méthodes qui font référence à un indice sont exécutées en temps O(n).

java.util.Vector :

La classe java.util.Vector est une classe héritée de Java 1. Elle n'est conservée dans l'API actuelle que pour des raisons de compatiblité ascendante et elle ne devrait pas être utilisée dans les nouveaux programmes.

Dans tous les cas, il est préférable d'utiliser un ArrayList.

Note : Cette classe est "thread-safe", c'est-à-dire que plusieurs processus peuvent l'utiliser en même temps sans risque.

Complexité : idem que pour ArrayList, plus le temps de synchronisation des méthodes.

Les constructeurs

LinkedList()

crée une liste chaînée vide.

LinkedList(Collection c)

crée une liste chaînée contenant les éléments de la collection passée en argument.

Méthodes

void add(int index, Object element)
insère l'élément spécifié à la position donnée au sein de l''objet *LinkedList*.
boolean add(Object o)
ajoute l'élément spécifié à la fin de l'objet *LinkedList*.
boolean addAll(Collection c)
ajoute tous les éléments de la collection spécifiée à la fin de la liste chaînée.
boolean addAll(int index, Collection c)
insère à partir de la position donnée, tous les les éléments de la collection spécifiée au sein de l'objet *LinkedList*.
void addFirst(Object o)
insère l'élément donné au début de la liste chaînée.
void addLast(Object o)
ajoute l'élément donné à la fin de la liste chaînée.
void clear()
supprime tous les éléments de l'objet *LinkedList*.
Object clone()
retourne une copie de référence de l'objet *LinkedList*.
boolean contains(Object o)
retourne *true* si la liste chaînée contient l'élément spécifié.
Object get(int index)
retourne l'élément trouvé à la position spécifiée.

Object getFirst()
retourne le premier élément de la liste chaînée.
Object getLast()
retourne le dernier élément de la liste chaînée.
int indexOf(Object o)
retourne l'index de la première occurrence de l'élément spécifié, ou -1 si ce dernier n'est pas trouvé.
int lastIndexOf(Object o)
retourne l'index de la dernière occurrence de l'élément spécifié, ou -1 si ce dernier n'est pas trouvé.
ListIterator listIterator(int index)
retourne un objet *ListIterator* contenant les éléments de la liste chaînée, à partir de l'index spécifié.
Object remove(int index)
supprime l'élément trouvé à la position spécifié au sein de la liste chaînée.
boolean remove(Object o)
supprime la première occurrence de l'élément spécifié.
Object removeFirst()
supprime et retourne le premier élément de la liste chaînée.
Object removeLast()
supprime et retourne le dernier élément de la liste chaînée.
Object set(int index, Object element)
remplace l'élément situé à la position spécifiée par l'élément passé en argument.
int size()
retourne le nombre d'éléments contenus dans la liste chaînée.
Object[] toArray()
retourne un tableau contenant tous les éléments de la liste chaînée dans un ordre exact.
Object[] toArray(Object[] a)
retourne un tableau contenant tous les éléments de la liste chaînée dans un ordre exact. Le type d'exécution du tableau retourné est celui du tableau passé en argument.

Exemple

```
import java.util.*;

public class LinkedListDemo{
  public static void main(String[] args){
    LinkedList liste=new LinkedList();
    liste.add("a");
    liste.add("b");
    liste.add(new Integer(10));
    System.out.println("Le contenu de la liste est : " + liste);
    System.out.println("La taille est : " + liste.size());

    liste.addFirst(new Integer(20));
    System.out.println("Le contenu de la liste est : " + liste);
    System.out.println("La taille est : " + liste.size());

    liste.addLast("c");
    System.out.println("Le contenu de la liste est : " + liste);
    System.out.println("La taille est : " + liste.size());

    liste.add(2,"j");
    System.out.println("Le contenu de la liste est : " + liste);
    System.out.println("La taille est : " + liste.size());

    liste.add(1,"t");
    System.out.println("Le contenu de la liste est : " + liste);
    System.out.println("La taille est : " + liste.size());

    liste.remove(3);
    System.out.println("Le contenu de la liste est : " + liste);
    System.out.println("La taille est : " + liste.size());
  } }
```

```
Le contenu de la liste est : [a, b, 10]
La taille est : 3

Le contenu de la liste est : [20, a, b, 10]
La taille est : 4

Le contenu de la liste est : [20, a, b, 10, c]
La taille est : 5

Le contenu de la liste est : [20, a, j, b, 10, c]
La taille est : 6

Le contenu de la liste est : [20, t, a, j, b, 10, c]
La taille est : 7

Le contenu de la liste est : [20, t, a, b, 10, c]
La taille est : 6
```

Exemple pour ArrayList

Les opérations principales sur un ArrayList sont :

add(Object o) :	ajoute l'objet o à la fin du ArrayList
clear() :	vide le ArrayList
get(int index) :	renvoie l'Object à l'index spécifié. Renvoie une exception si vous dépassez le tableau (IndexOutOfBoundsException)
size() :	renvoie la taille du ArrayList

```
// on crée un ArrayList de taille 10
List monArrayList = new ArrayList(10) ;

// on ajoute 30 entiers
for(int i=0;i<30;i++) {
    monArrayList.add(new Integer(i));
}
```

On remarque que le ArrayList , au moment d'ajouter 10, atteint sa capacité maximale, il se redimensionne pour en accepter plus

Types Génériques

Permet de spécifier des paramètres dénotant des types à utiliser dans une classe

Exemples classiques : les collections

- Collection<E>
- Set<E>
- List<E>
- Map<K,V>

29

Exemples d'usage de types génériques

```
import java.util.*;
public class Test
{
  public static void main(String[] args)
  {
  Set<Integer> m = new HashSet<Integer>();
  m.add(new Integer(1));
  for (Integer i : m)
    System.out.println(i);
  }
}
```

30