Structures collectives en Java

Array – List - Map

Motivations

- 1, 2, … plusieurs
  - monôme, binôme, … polynôme
  - point, segment, triangle, … polygone
- Importance en conception
  - Relation entre classes
    - «… est une collection de … »
    - « … a pour composant une collection de … »
  - Choisir la meilleure structure collective
    - plus ou moins facile à mettre en œuvre
    - permettant des traitements efficaces
    - selon la taille (prévue) de la collection

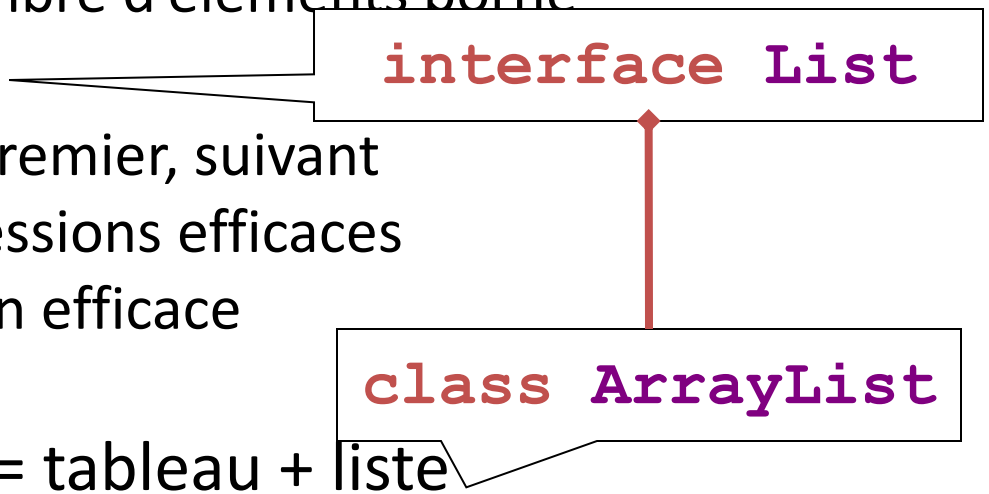
Définition d ’une collection

- Une ***collection*** regroupe plusieurs données de même nature
    - Exemples : promotion d’étudiants, sac de billes, ...
- 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, …)

Structures collectives classiques

`type[]` et `Array`

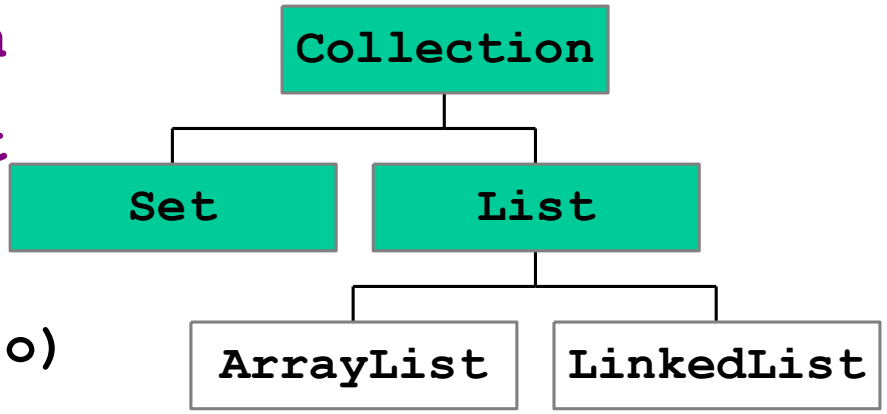
- Tableau
  - accès par index
  - recherche efficace si le tableau est trié (dichotomie)
  - insertions et suppressions peu efficaces
  - défaut majeur : nombre d'éléments borné
- Liste
  - accès séquentiel : premier, suivant
  - insertions et suppressions efficaces
  - recherche lente, non efficace



- Tableau dynamique = tableau + liste

Paquetage `java.util` de Java 2

- 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

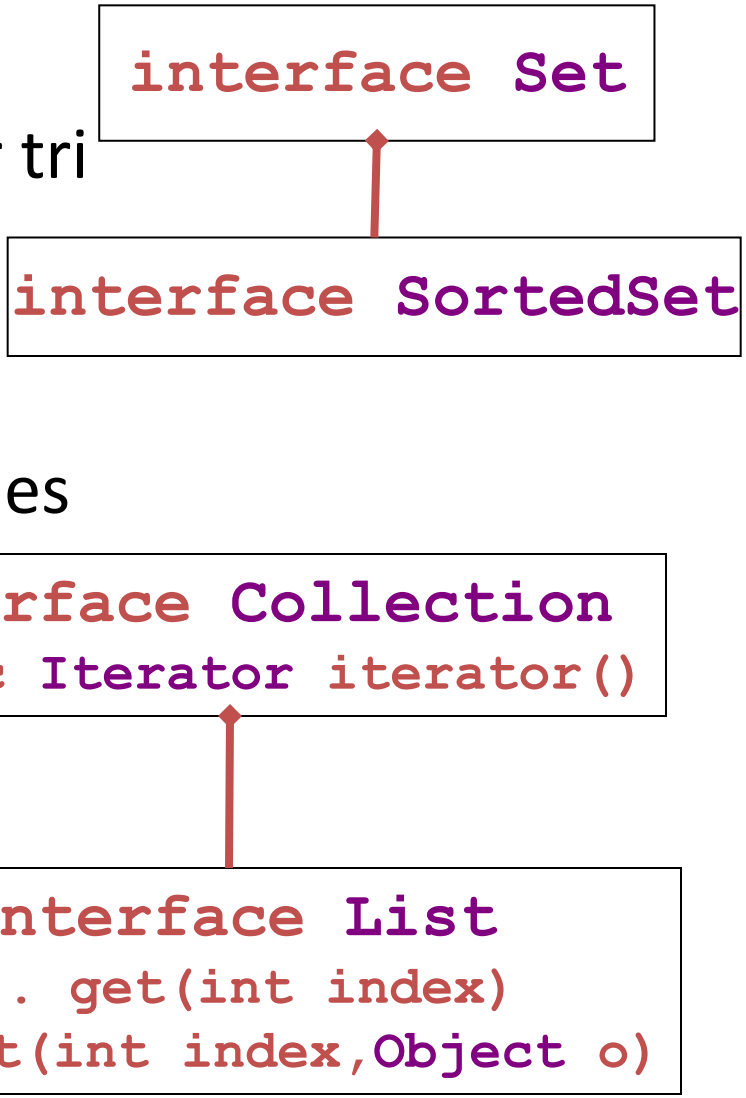
`boolean add(Object)` : ajouter un élément
`boolean addAll(Collection)` : ajouter plusieurs éléments
`void clear()` : tout supprimer
`boolean contains(Object)` : test d'appartenance
`boolean containsAll(Collection)` : appartenance collective
`boolean isEmpty()` : test de l'absence d'éléments
`Iterator iterator()` : pour le parcours (cf Iterator)
`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()`, `removeFisrt()`, `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)

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`**

Généricité des algorithmes

- N'utiliser que les méthodes communes
- Déclaration
  - `Collection col = new ArrayList();`
- Parcours des éléments de la collection : **`Iterator`**
  - accès indexé pas toujours disponible (méthode **`get()`**)
  - utiliser la méthode **`Iterator iterator()`**
  - se déplacer avec les méthodes **`next()`** et **`hasNext()`**
  - exemple

```
Collection col = new TreeSet();
Iterator i = col.iterator();
while (i.hasNext())
      traiter
((transtypage)i.next());
```

Vue d'ensemble des collections

- Hiérarchie simplifiée