Collections

Université Française d'Égypte
Version O 0.9 – 10/10/12
Richard Grin

Plan du cours

- Généralités sur les collections
- Collections et itérateurs
- Maps (« collections » indexées par des clés)
- Utilitaires : trier une collection et rechercher une information dans une liste triée

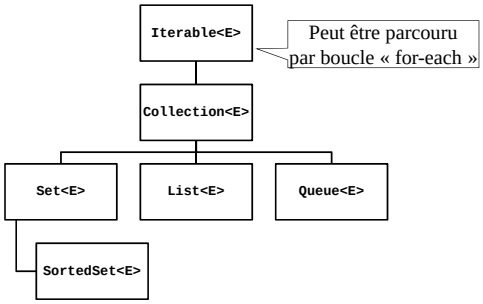
Définition d'une collection

- Objet dont la principale fonctionnalité est de contenir d'autres objets
- Divers types de collections dans le paquetage **`java.util`**

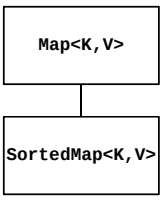
Généricité

- A partir du JDK 5, la généricité permet d'indiquer le type des objets contenus dans une collection :
  `List<Employe>`

Hiérarchie des interfaces - `Collection`



Hiérarchie des interfaces – `Map`



Classes concrètes d'implantation des interfaces

		Classes d'implantation			
		Table de hachage	Tableau	Arbre balancé	Liste chaînée
Interfaces	Set<E>	HashSet<E>		TreeSet<E>	
	List<E>		ArrayList<E>		LinkedList<E>
	Map<K,V>	HashMap<K,V>		TreeMap<K,V>	

2 exemples d'introduction

Exemple de liste

```
List<String> l = new ArrayList<String>();
l.add("Pierre Jacques");
l.add("Pierre Paul");
l.add("Jacques Pierre");
Collections.sort(l);
System.out.println(l);
```


Exemple de liste

```
List<String> l = new ArrayList<>();
l.add("Pierre Jacques");
l.add("Pierre Paul");
l.add("Jacques Pierre");
Collections.sort(l);
System.out.println(l);
```


Exemple de Map

```
Map<String, Integer> frequences =
  new HashMap<String, Integer>();
for (String  mot : args) {
  Integer freq = frequences.get(mot);
  if (freq == null)
    freq = 1;
  else
    freq++;
  frequences.put(mot, freq);
}
System.out.println(frequences);
```


Exemple de Map

```
Map<String, Integer> frequences =
  new HashMap<>();
for (String  mot : args) {
  Integer freq = frequences.get(mot);
  if (freq == null)
    freq = 1;
  else
    freq++;
  frequences.put(mot, freq);
}
System.out.println(frequences);
```

Collections et types primitifs

- Les collections ne peuvent contenir de valeurs des types primitifs
- Avant le JDK 5, il fallait utiliser les classes enveloppantes des types primitifs, **`Integer`** par exemple
- A partir du JDK 5, le « boxing » / « unboxing » simplifie l'écriture

Exemple de liste avec boxing

```
List<Integer> l = new ArrayList<>();
l.add(10);
l.add(-678);
l.add(87);
l.add(7);
int i = l.get(0);
```


Interfaces **`Iterator<E>`** et **`Iterable<E>`**

`Iterator<E>`

- Permet d'énumérer les éléments d'une collection
- Encapsule la structure de la collection

Obtenir un itérateur

- Méthode de **`Collection<E>`** **`Iterator<E> iterator()`** renvoie un itérateur de la collection
- **`List<E>`** contient en plus la méthode **`ListIterator<E> listIterator()`** qui renvoie un **`ListIterator`** (plus de possibilités pour parcourir une liste et la modifier)

Méthodes de l'interface **`Iterator<E>`**

```
boolean hasNext()
E next()
* void remove()
```

- **`remove()`** enlève le dernier élément récupéré par l'itérateur

Itérateur et modification de la collection parcourue

1. On obtient un itérateur pour une collection
2. On modifie la collection directement (sans passer par l'itérateur)
3. Utiliser l'itérateur lance une **`ConcurrentModificationException`**

Exemple d'utilisation de **`Iterator`**

```
List<Employe> l = new ArrayList<Employe>();
Employe e = new Employe("Dupond");
l.add(e);
// ajoute d'autres employés dans l
. . .
Iterator<Employe> it = l.iterator();
while (it.hasNext()) {
  // le 1er next() fournit le 1er élément
  System.out.println(it.next().getNom());
}
```


Interface **`Iterable<T>`**

- Indique qu'un objet peut être parcouru par un itérateur :

```
public interface Iterable<T>  {
  Iterator<T> iterator();
}
```

- Toute classe qui implémente **`Iterable`** peut être parcourue par une boucle « for each »
- L'interface **`Collection`** en hérite

Boucle « for each »

- Boucle « normale » :

```
for (Iterator<Employe> it = coll.iterator();
     it.hasNext(); ) {
  Employe e = it.next();
  String nom = e.getNom();
}
```

- Avec une boucle « for each » :

```
for (Employe e : coll) {
  String nom = e.getNom();
}
```


Restriction de « for each »

- On ne dispose pas de la position dans la collection pendant le parcours
- On ne peut pas modifier la collection pendant le parcours de la boucle (alors que c'est possible par l'intermédiaire de l'itérateur)
- Boucle ordinaire avec itérateur indispensable si ces 2 restrictions gênent

Principes généraux sur les déclarations de types pour favoriser la réutilisation

Principe général pour la réutilisation

- Le code doit fournir ses services au plus grand nombre possible de clients
- Les conditions d'utilisation des méthodes doivent être les moins contraignantes possible

Paramètres des méthodes

- Il vaut mieux déclarer les paramètres du type **interface** le plus général *possible* :
  - **`m(Collection)`** plutôt que **`m(List)`**
  - éviter **`m(ArrayList)`**
- On élargit ainsi le champ d'utilisation de la méthode

Type retour des méthodes (1)

- On peut déclarer le type retour du type le plus spécifique possible **si ce type ajoute des fonctionnalités** :
  « **`List m()`** » plutôt que « **`Collection m()`** »
- L'utilisateur de la méthode
  - peut ainsi profiter de toutes les fonctionnalités offertes par le type de l'objet retourné
  - mais rien ne l'empêche de faire un « *upcast* » avec l'objet retourné : **`Collection l = m(...)`**

Type retour des méthodes (2)

- Mais il faut être certain que l'instance retournée par la méthode sera toujours du type déclaré
- Problème si on déclare que le type retour est de type **`List`** mais qu'on souhaite plus tard renvoyer un **`Set`**

Variables

- Les collections sont déclarées du type d'une interface :

```
List<Employe> employes =
  new ArrayList<Employe>();
```

- Il sera ainsi possible de changer d'implémentation ailleurs dans le code :

```
employes =
  new LinkedList<Employe>();
```


Tri et recherche dans une collection

Classe **`Collections`**

- Contient des méthodes **`static`**, pour travailler avec des collections :
  - tris (sur listes)
  - recherches (sur listes triées)
  - minimum et maximum
  - ...

Trier une liste

- **`Collections.sort(liste);`**
- Cette méthode ne renvoie rien ; elle trie **`liste`**
- Les éléments de la liste doivent implémenter l'interface **`java.lang.Comparable<T>`** pour un sur-type **`T`** du type **`E`** de la collection

Interface **`Comparable<T>`**

- Ordre « naturel » dans les instances d'une classe
- Une seule méthode :
  `int compareTo(T t2)`
  qui renvoie
  - un entier positif si **`this`** est plus grand que **`t2`**
  - 0 si les 2 objets ont la même valeur
  - un entier négatif sinon

Classes qui implantent **`Comparable`**

- Classes du JDK qui enveloppent les types primitifs (**`Integer`**,…)
- Classes **`String`**, **`Date`**, **`Calendar`**, **`BigInteger`**, **`BigDecimal`** et quelques autres
- Par exemple, **`String`** implémente **`Comparable<String>`**

Question

- Que faire
  - si les éléments de la liste n'implantent pas l'interface **`Comparable`**,
  - ou si on veut les trier suivant un autre ordre que celui donné par **`Comparable`** ?

Réponse

1. On construit un objet qui sait comparer 2 éléments de la collection (interface **`java.util.Comparator<T>`**)
2. On passe cet objet en paramètre à la méthode **`sort`**

Interface **`Comparator<T>`**

- Une seule méthode :
  `int compare(T t1, T t2)`
  qui renvoie
  - un entier positif si **`t1`** est « plus grand » que **`t2`**
  - 0 si **`t1`** a la même valeur (au sens de **`equals`**) que **`t2`**
  - un entier négatif sinon

Exemple de comparateur

```
public class CompareSalaire
    implements Comparator<Employe> {
  public int compare(Employe e1, Employe e2) {
    double s1 = e1.getSalaire();
    double s2 = e2.getSalaire();
    if (s1 > s2)
      return +1;
    else if (s1 < s2)
      return –1;
    else
      return 0;
  }
}
```


Utilisation d'un comparateur

```
List<Employe> employes = new ArrayList<>();
// On ajoute les employés
. . .
Collections.sort(employes,
                 new CompareSalaire());
System.out.println(employes);
```