Collections

Université Française d'Egypte
Version O 7.0.1 – 5/7/13
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

- Permet d'indiquer le type des objets contenus dans une collection :
  `List<Employe>`

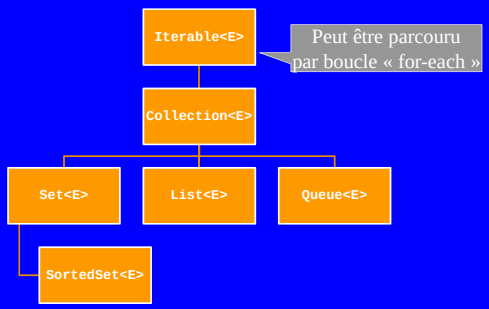
Les interfaces

- 2 hiérarchies d'héritage principales :
  - `Collection<E>` : collections proprement dites
  - `Map<K,V>` : collections indexées par des clés

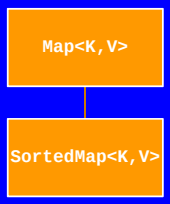
Type des clés

Type des valeurs indexées par les clés

Hiérarchie des interfaces - `Collection`



Hiérarchie des interfaces – `Map`



Les classes abstraites

- **`AbstractCollection<E>`**, **`AbstractList<E>`**, **`AbstractMap<K,V>`**,… implantent les méthodes communes aux différents types de collection

Les classes concrètes

- **`ArrayList<E>`**, **`LinkedList<E>`**, **`HashMap<K,V>`**, **`TreeMap<K,V>`**,… héritent des classes abstraites
- Implantent le support concret qui va recevoir les objets des collections (tableau, table de hachage, liste chaînée,…)

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);
```


Interface **`Collection<E>`**

Méthodes de `Collection<E>`

```
* boolean add(E elt)
* void addAll(Collection<? extends E> c)
* void clear()
boolean contains(Object obj)
* boolean remove(Object obj)
int size()
Object[] toArray()
<T> T[] toArray(T[] tableau)
```

* : méthode optionnelle (pas nécessairement disponible)

Méthode optionnelle

- Nombreux cas particuliers de collections :
  - collections de taille fixe
  - collections dont on ne peut enlever des objets
  - …
- Plutôt que de fournir une interface pour chaque cas particulier, l'API comporte la notion de méthode optionnelle

Méthode optionnelle

- Méthode qui peut renvoyer une `java.lang.UnsupportedOperationException` dans une classe qui ne la supporte pas

Convention sur les constructeurs

- Toute classe d'implantation des collections doit fournir au moins 2 constructeurs :
  - un constructeur sans paramètre
  - un constructeur qui prend une collection quelconque d'éléments de type compatible en paramètre ; par exemple : `ArrayList(Collection<? extends E> c)`

Interface `Set<E>`

Définition de l'interface `Set<E>`

- Collection qui ne contient pas 2 objets égaux au sens de `equals`
- Rappel : si `equals` est redéfini dans `E`, `hashcode` doit aussi l'être

Méthodes de `Set<E>`

- Mêmes méthodes que l'interface **`Collection`**
- Mais les « contrats » des méthodes sont adaptés aux ensembles
- Par exemple, la méthode **`add`** n'ajoute pas un élément si un élément égal (au sens de **`equals`**) est déjà dans l'ensemble

Implémentation

- **`HashSet<E>`** garantit un temps constant pour les opérations de base (**`set`**, **`add`**, **`remove`**, **`size`**)
- **`TreeSet<E>`** garantit que les éléments sont rangés dans un certain ordre

Interface `List<E>`

Définition de `List<E>`

- Collection d'objets indexés par des numéros (en commençant par 0)

Implémentations

- **`ArrayList<E>`** tableau à taille variable
- **`LinkedList<E>`** liste chaînée

- On utilise le plus souvent **`ArrayList`**, sauf si les insertions/suppressions au milieu de la liste sont fréquentes

Nouvelles méthodes de `List<E>`

```
* void add(int indice, E elt)
* boolean addAll(int indice,
                 Collection<? extends E> c)
E get(int indice)
* E set(int indice, E elt)
* E remove(int indice)
int indexOf(Object obj)
int lastIndexOf(Object obj)
List<E> subList(int depuis, int
```

insertion avec décalage vers la droite des indices

suppression avec décalage vers la gauche des indices

indice du 1er élément égal à **`obj`** (au sens de **`equals`**) (ou -1)

Ne pas oublier !

- On peut aussi utiliser toutes les méthodes héritées de **`Collection`**, en particulier
  - **`remove(Object)`**
  - **`add(E)`**

Classe `ArrayList<E>`

Exemple d'utilisation de `ArrayList`

```
List<Employe> le = new ArrayList<>();
Employe e = new Employe("Dupond");
le.add(e);
// Ajoute d'autres employés
. . .
for (int i = 0; i < le.size(); i++) {
  System.out.println(le.get(i).getNom());
}
```

```
for (Employe e : le) {
  System.out.println(e.getNom());
}
```


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

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());
}
```


Itérateur et modification de la collection parcourue

1. Récupérer un itérateur pour une collection
2. Modifier la collection directement (sans passer par l'itérateur)
3. Utiliser l'itérateur lance alors une **`ConcurrentModificationException`**

Itérateur de liste

- L'interface **`ListIterator`** permet
  - de parcourir la liste sous-jacente dans les 2 sens
  - de modifier cette liste avec les méthodes *optionnelles* **`add`** et **`set`**

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

Interface `Map<K,V>`

Cas d'utilisation

- Ranger et rechercher une valeur identifiée par une clé
- Exemple : rechercher les informations sur un étudiant identifié par son numéro d'étudiant

Définition de `Map<K,V>`

- Couples clé - valeur
- Une clé repère une et une seule valeur
- Il ne peut exister 2 clés égales au sens de `equals`

Implémentation

- `HashMap<K,V>`, table de hachage ; accès en temps constant
- `TreeMap<K,V>`, arbre ordonné suivant les valeurs des clés ; accès en log(n)

Fonctionnalités

- On peut
  - ajouter et enlever des couples clé – valeur
  - récupérer une des valeurs en donnant sa clé
  - savoir si une table contient une valeur
  - savoir si une table contient une clé

Méthodes de `Map<K,V>`

```
* void clear()
boolean containsKey(Object clé)
boolean containsValue(Object valeur)
V get(Object clé)
boolean isEmpty()
Set<K> keySet()
Collection<V> values()
Set<Map.Entry<K,V>> entrySet()
* V put(K clé, V valeur)
* void putAll(Map<? extends K, ? extends V>
  map)
* V remove(Object key)
int size()
```

retourne `null` si la clé n'existe pas

Interface *interne* `Entry<K,V>` de `Map`

- Interface interne **`public`** qui correspond à un couple clé - valeur
- 3 méthodes
  - `K getKey()`
  - `V getValue()`
  - `V setValue(V valeur)`
- La méthode **`entrySet()`** de **`Map`** renvoie un objet de type « ensemble (**`Set`**) de **`Entry`** »

Constructeurs

- Par convention toute classe d'implantation de **`Map`** doit fournir au moins 2 constructeurs :
  - un constructeur sans paramètre
  - un constructeur qui prend une *map* de type compatible en paramètre

Modification des clés

- Si on veut changer une clé,
  1. on enlève d'abord l'ancienne entrée
  2. on ajoute ensuite la nouvelle entrée avec la nouvelle clé et l'ancienne valeur

Récupérer les valeurs d'une `Map`

1. Méthode **`values()`** renvoie une `Collection<V>` (qui reflétera les modifications futures de la *map*, et vice-versa)
2. Méthode **`iterator()`** de l'interface **`Collection<V>`** pour récupérer un à un les éléments

Récupérer les clés d'une `Map`

1. Méthode **`keySet()`** qui renvoie un **`Set<K>`**
2. Méthode **`iterator()`** de l'interface **`Set<K>`** pour récupérer une à une les clés

Récupérer les entrées d'une `Map`

1. Méthode `entrySet()` qui renvoie un `Set<Entry<K,V>>`
2. Méthode `iterator()` de l'interface `Set<Entry<K,V>>` pour récupérer une à une les entrées

Classe `HashMap<K,V>`

Implémentation

- `HashMap<K,V>` utilise une table de hachage pour ranger les clés

Exemple d'utilisation de `HashMap`

```
Map<String,Employe> hm = new HashMap<>();
Employe e = new Employe("E125","Dupond");
...
hm.put(e.getMatricule(), e);
// Ajoute d'autres employés dans la table de hachage
. .
Employe e2 = hm.get("E369");
Collection<Employe> employes =
  hm.values();
for (Employe employe : employes) {
  System.out.println(employe.getNom());
}
```


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 ne 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

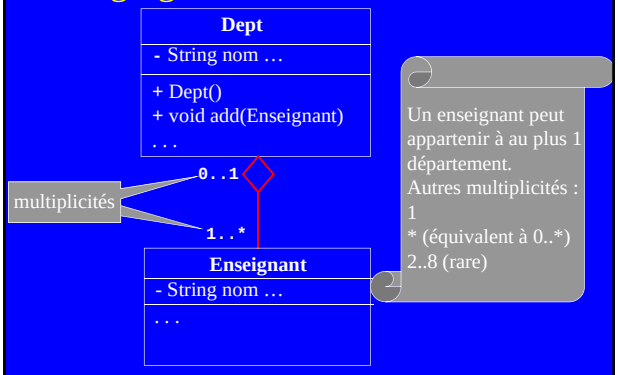
Exemple de comparateur

```
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);
```


L'agrégation en notation UML



Compatibilité avec les classes fournies par le JDK 1.1

Classes du JDK 1.1

- Dans les API du JDK 1.1, on utilisait
  - la classe **`Vector`** (remplacée par **`ArrayList`**)
  - la classe **`Hashtable`** (remplacée par **`HashMap`**)
  - l'interface **`Enumeration`** (remplacée par **`Iterator`**)
- Il vaut mieux utiliser les nouvelles classes