Collections

Université Française d'Egypte
Version O 8.15 – 13/12/20
Richard Grin

1

Plan du cours

- Généralités sur les collections
- Exemples d'introduction
- Collections
- Ensembles
- Listes
- Itérateurs
- Maps (« collections » indexées par des clés)
- Faciliter la réutilisation
- Tri et recherche dans une liste

2

Définition d'une collection

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

3

Généricité

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

4

Les interfaces

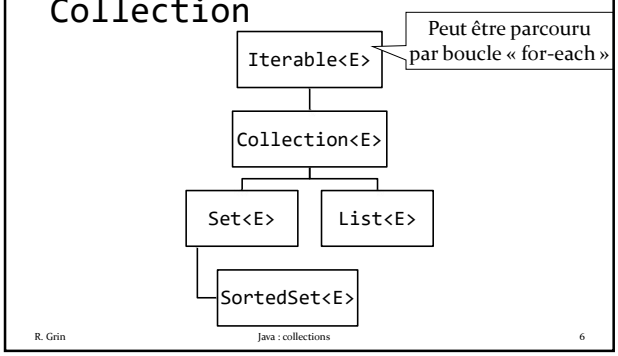
- 2 hiérarchies d'héritage principales :
  - `Collection<E>` : collections proprement dites
  - `Map<K,V>` : collections indexées par des clés

Type des clés

Type des valeurs indexées par les clés

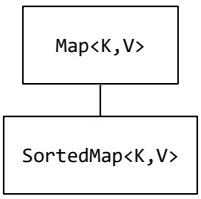
5

Hiérarchie des interfaces – `Collection`



6

Hiérarchie des interfaces – Map

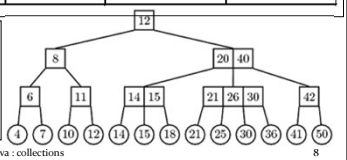


7

Classes concrètes qui implémentent les interfaces

		Classes d'implémentation			
		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>	

Vous connaissez déjà les tableaux, les listes chaînées, les tables de hachage.
Voici l'image d'un arbre :



8

Collections et types primitifs

- Une collection ne peut pas contenir une valeur de type primitif
- Le « boxing » / « unboxing » simplifie l'écriture

9

Exemple de liste avec (un)boxing

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

Comment l'écrire sans (un)boxing ?

10

Exemple de liste sans (un)boxing

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


11

2 exemples d'introduction

12

Exemple avec liste

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


13

Exemple avec liste

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

`sort(null)` : trier par l'ordre naturel

Que fait ce code ?

Affiche [Jacques Pierre, Pierre Jacques, Pierre Paul].
Qu'est-ce que ça suppose sur la classe `ArrayList` ?

14

Exemple avec Map

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


15

Exemple avec Map

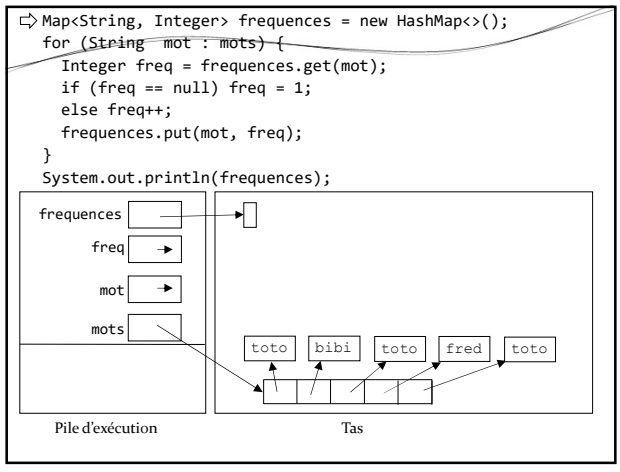
```
Map<String, Integer> frequences =
    new HashMap<>();
for (String  mot : mots) {
  Integer freq = frequences.get(mot);
  if (freq == null)
    freq = 1;
  else
    freq++;
  frequences.put(mot, freq);
}
System.out.println(frequences);
```

`mots` est de type `String[]`

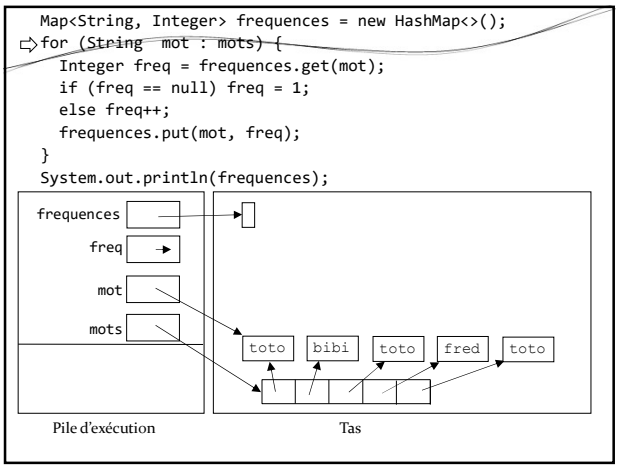
Que fait ce code ?

Aide : affiche {toto=3, bibi=1, fred=1}
si `mots` contient ["toto", "bibi", "toto", "fred", "toto"]

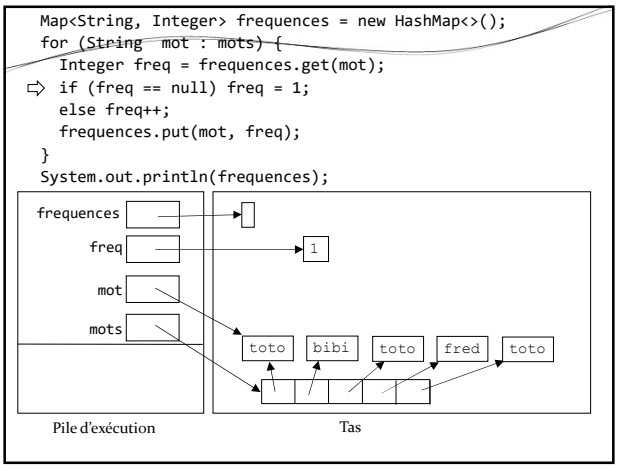
16



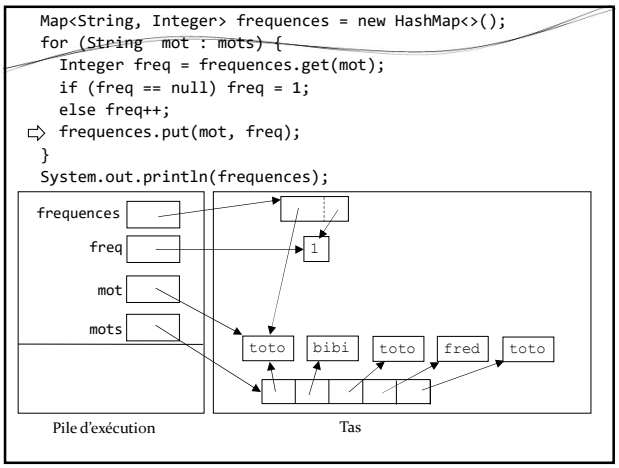
17



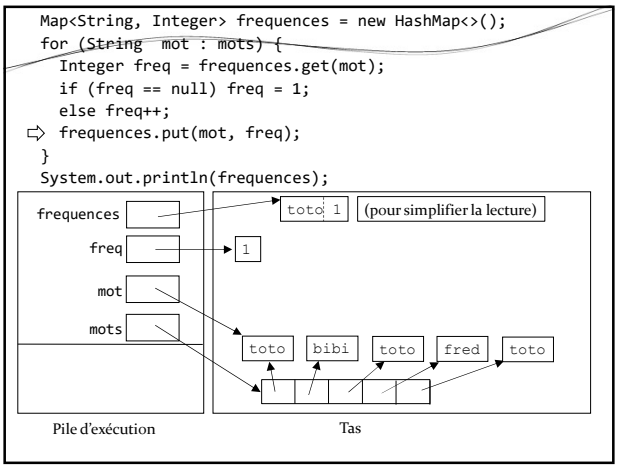
18



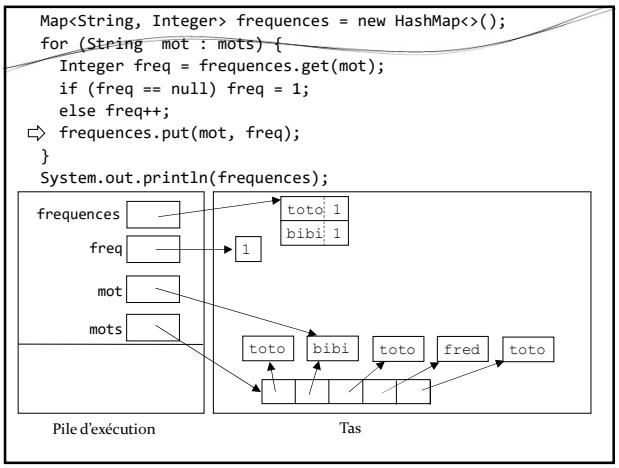
19



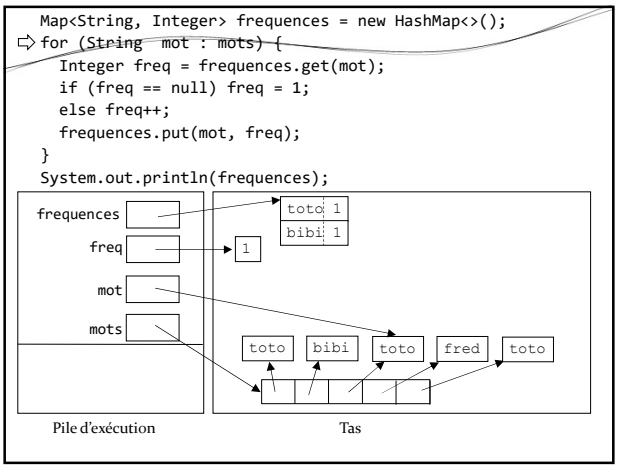
20



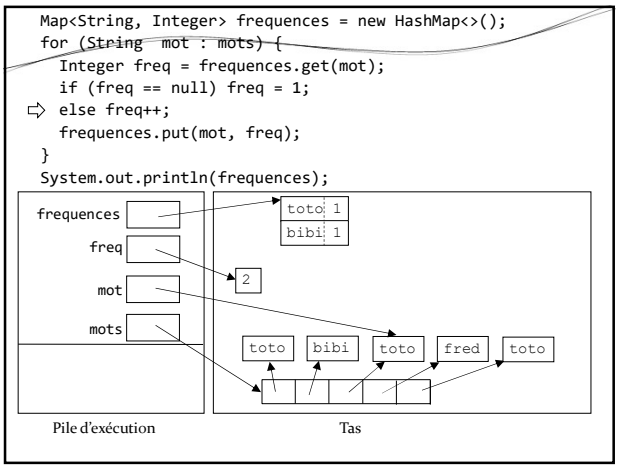
21



22

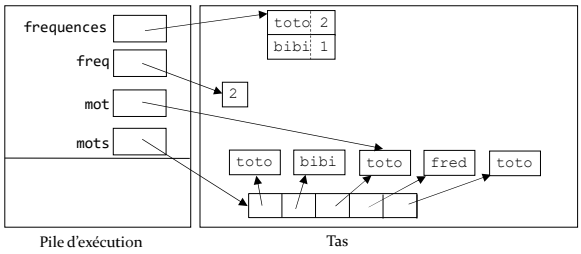


23



24

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



25

Interface Collection<E>

26

Méthodes de Collection<E>

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

* : méthode optionnelle (pas nécessairement disponible)

27

Méthode optionnelle

- Nombreux cas particuliers de collections :
  - collection de taille fixe
  - collection dont on ne peut enlever des objets
  - …
- Plutôt que de fournir une interface pour chaque cas particulier, l'API a la notion de méthode optionnelle : une méthode peut lancer une `UnsupportedOperationException` si la collection ne supporte pas l'opération

Pas une bonne pratique de conception mais, pour ce cas, c'est un bon compromis

28

Convention sur les constructeurs

- Toute classe d'implémentation 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, dans `ArrayList<E>` :
    `ArrayList(Collection<? extends E> c)`
    Ainsi, dans une collection de `Personne`, on peut ajouter tous les éléments d'une collection de `Client`, si `Client` hérite de `Personne`

29

Interface Set<E>

30

Définition de `Set<E>`

- Comme les ensembles en mathématiques : collection qui ne contient pas 2 objets égaux au sens de `equals`
- En mathématiques,
  { a, b, c } ∪ { a, d } = { a, b, c, d }

31

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

32

Implémentations

- `HashSet<E>` temps constant pour les opérations de base (`add`, `remove`, `size`)
- `TreeSet<E>` éléments rangés dans un certain ordre (implémente `SortedSet`)

33

Interface `List<E>`

34

Définition de `List<E>`

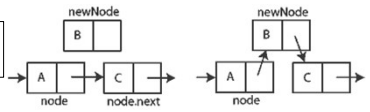
- Collection d'objets indexés par des numéros (en commençant par 0)

35

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

Insertion avec liste chaînée :



36

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 jusqua)
```

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`

37

`E get(int indice)`

- Lance une `IndexOutOfBoundsException` si l'indice est < 0 ou >= `size()`
- `IndexOutOfBoundsException` est une exception *runtime* du paquetage `java.lang`

Signification ?

38

Ne pas oublier !

- On peut aussi utiliser toutes les méthodes héritées de `Collection<E>`, en particulier
  - `boolean remove(Object)`
  - `boolean add(E)`
  - `int size()`

Attention, si on veut enlever un entier d'une liste d'entiers, il faut le caster avec `Integer` pour que le compilateur ne confonde pas avec `remove(int indice)` de l'interface `List<E>`

39

Exemple 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());
}
```

Boucle « for-each » qui utilise un itérateur

40

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

41

`Iterator<E>`

- Permet d'énumérer les éléments d'une collection (pas d'autres façons pour `Set<E>` ou d'autres collections)
- Encapsule la structure de la collection (même code si la collection est une liste ou un ensemble)

42

Méthodes de l'interface `Iterator<E>`

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

- `remove()` enlève le dernier élément récupéré par `next()` (méthode optionnelle)

Obtenir un itérateur

- Méthode de `Collection<E>`
  `Iterator<E> iterator()`
  retourne un itérateur de la collection

Exemple 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 de liste

- `List<E>` contient la méthode
  `ListIterator<E> listIterator()`
- L'interface `ListIterator` hérite de `Iterator`
- En plus elle permet de
  - parcourir la liste dans les 2 sens
  - modifier cette liste avec les méthodes *optionnelles* `add` (insertion d'un élément) et `set` (remplacement d'un élément)

Itérateur et modification de la collection parcourue

1. On récupère un itérateur pour une collection
2. On modifie la collection directement (sans passer par l'itérateur)
3. Appeler les méthodes `next` ou `remove` de l'itérateur lance alors une `ConcurrentModificationException`

Donc, si on veut utiliser un itérateur pour récupérer tous les éléments de la collection, il ne faut pas modifier cette collection sans passer par l'itérateur

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<Client> it = clients.iterator();
     it.hasNext(); ) {
  Client client = it.next();
  String nom = client.getNom();
  ...
}
```

- Avec une boucle « for each » :

```
for (Client client : clients) {
  String nom = client.getNom();
  ...
}
```


Restriction boucle « for each »

- On n'a pas la position dans une liste pendant son parcours
- La collection ne peut pas être modifiée pendant le parcours de la boucle (alors que c'est possible par l'intermédiaire de l'itérateur)
- Une boucle ordinaire avec itérateur est indispensable si ces 2 restrictions gênent

Méthode `forEach` de `Iterable`

- `default void forEach(Consumer<? super E> action)` applique `action` à chaque élément itéré
- Exemple : Expression lambda
  `liste.forEach(x -> System.out.println(x))`
- qui peut aussi s'écrire
  `liste.forEach(System.out::println)`
  Référence de méthode

Interface `Map<K,V>`

Cas d'utilisation

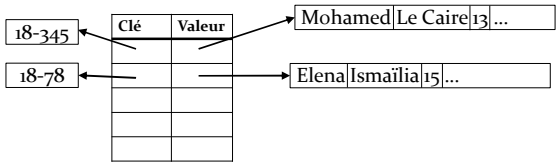
- Ranger et rechercher une valeur identifiée par une clé
- Exemple : rechercher les informations sur un étudiant identifié par son numéro d'étudiant

Définition de `Map<K,V>`

- Couples (clé (`K`), valeur (`V`))
- Une clé donne accès rapidement à une valeur
- Il ne peut pas exister 2 clés égales au sens de `equals`

Exemple

- Une map qui contient des étudiants indexés par leur numéro d'étudiant



55

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)

56

Fonctionnalités

- On peut
  - ajouter et enlever des couples clé – valeur
  - récupérer une des valeurs en donnant sa clé
  - savoir si une table contient une clé (très rapide)
  - savoir si une table contient une valeur (peut être long)

57

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 clé n'existe pas

Pourquoi des types retour différents ?

58

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` »

Pourquoi « ensemble » et pas « collection » ?

59

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

60

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

61

Récupérer les valeurs d'une Map

1. Méthode `values()` retourne une `Collection<V>`
2. Méthode `iterator()` de l'interface `Collection<V>` pour récupérer un à un les éléments (ou bien une boucle for-each)

62

Récupérer les clés d'une Map

1. Méthode `keySet()` retourne un `Set<K>`
2. Méthode `iterator()` de l'interface `Set<K>` pour récupérer une à une les clés (ou bien une boucle for-each)

63

Exemple d'utilisation de HashMap

```
Map<String,Employe> hm = new HashMap<>();
Employe e = new Employe("E125","Dupond");
hm.put(e.getMatricule(), e);
...
Employe e2 = hm.get("E369");
if (e2 != null) { ... } else { ... }
...
Collection<Employe> employes = hm.values();
for (Employe employe : employes) {
    System.out.println(employe.getNom());
}
```


64

Complément sur Map<K,V>

- `default void forEach(BiConsumer<? super K, ? super V> action)`
  applique `action` à chaque entrée de la map
- Exemple :

```
map.forEach((cle, valeur) ->
   System.out.print(cle + "-" + valeur))
```


65

Faciliter la réutilisation

66

Principe général pour la réutilisation

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

67

Paramètres des méthodes

- Si c'est possible, il vaut mieux déclarer un paramètre 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

68

Type retour des méthodes (1/2)

- Déclarer le type retour 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(...)`

69

Type retour des méthodes (2/2)

- 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 retourner un `Set`

70

Variables

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

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

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

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


71

Tri et recherche dans une liste

72

Classe `Collections`

- Contient des méthodes `static`, pour travailler avec des collections :
  - tris sur listes (`sort`)
  - recherches sur listes triées (`binarySearch`)
  - minimum et maximum (`min`, `max`)
  - …

73

Trier une liste

- `Collections.sort(liste);`
- Cette méthode ne retourne rien ; elle trie `liste`
- Les éléments de la liste doivent implémenter l'interface `java.lang.Comparable<T>` où `T` est un sur-type du type `E` de la collection
- Depuis Java 8 il est plus simple d'utiliser la méthode `sort` de `List` :
  `liste.sort(null);`

Que signifie « `T` est un sur-type de `E` » ?

74

Interface `Comparable<T>`

- Ordre « naturel » dans les instances de type `T`
- Si une classe implémente cette interface, indique qu'on peut comparer une instance de cette classe à un `T`
- Une seule méthode :
  `int compareTo(T t2)`
  qui retourne
  - un entier positif si `this` est plus grand que `t2`
  - 0 si les 2 objets ont la même valeur
  - un entier négatif sinon

75

Classes qui implémentent `Comparable`

- Classes du JDK qui enveloppent les types primitifs (`Integer`,…)
- Classes `String`
- Classes du JDK qui représente un moment ou une date (`Date`, `Calendar`, les classes de `java.time`)
- `BigInteger`, `BigDecimal`
- Par exemple, `String` implémente `Comparable<String>`

76

Exemple avec sur-type

- `Employé` hérite de `Personne`
- Soit une liste d'employé
- On sait trier la liste des employés si `Employé` implémente `Comparable<Employé>` ; `Comparable<Employé>` peut, par exemple, comparer les employés par leur salaire
- Mais on sait aussi trier la liste si `Employé` implémente `Comparable<Personne>` ; `Comparable<Personne>` peut, par exemple, comparer les personnes (donc les employés qui sont des personnes) par leur nom

77

Question

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

78

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` ; la méthode sort utilisera le `Comparator<T>` quand elle aura besoin de comparer 2 objets de type `T` pendant le tri

79

Interface `Comparator<T>`

- Une seule méthode :
  `int compare(T t1, T t2)`
  qui retourne
  - 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

80

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;
  }
}
```

Simplification si les salaires sont de type `int` ?

81

Utilisation d'un comparateur

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


82

`sort` de l'interface `List<E>`

- `default void sort(Comparator<? super E> comparateur)`
  trie la liste selon le comparateur (suivant l'ordre naturel de `E` si `null` est passé en paramètre)
  Ordre naturel. Qu'est-ce que c'est ?
- Exemple :

```
listeNoms.sort(
  String.CASE_INSENSITIVE_ORDER);
```

Dans quelle classe est défini `CASE_INSENSITIVE_ORDER` ?

Type de `CASE_INSENSITIVE_ORDER` ?

83

Compléments sur `Comparator` (1/2)

- Méthode `static comparing` qui retourne un `Comparator<T>` qui compare les `T` en utilisant une clé extraite des `T`
- Exemple avec `Comparator<Etudiant>` :
  `Comparator.comparing(e -> e.getPromo())`
- L'ordre naturel est utilisé pour comparer les clés (ici les promos de type `int`) ; passer un comparateur en 2ème paramètre si on veut un autre ordre

84

Compléments sur `Comparator` (2/2)

- Des méthodes `default` retournent des comparateurs liés au comparateur `this` :
- `Comparator<T> reversed()`
  le comparateur retourné inverse l'ordre de `this`
- `thenComparing` (3 méthodes surchargées) retourne un comparateur qui compare avec `this`, et une clé secondaire (donnée par les paramètres de la méthode)

85

Exemple

```
listeEtudiants.sort(
  Comparator.comparing((Etudiant e) -> e.getPromo())
  .reversed()
  .thenComparing(e -> e.getMoyenne()));
```

Que fait ce code ?

86

Comparer avec des types primitifs

- `comparingInt`, `comparingLong`, `comparingDouble` , `thenComparingInt`, `thenComparingLong` et `thenComparingDouble` sont plus efficaces quand la clé de tri est de type primitif

87

Exemple

```
listeEtudiants.sort(
  Comparator.comparingInt((Etudiant e) -> e.getPromo())
  .reversed()
  .thenComparingDouble(e -> e.getMoyenne()));
```

Que fait ce code ?

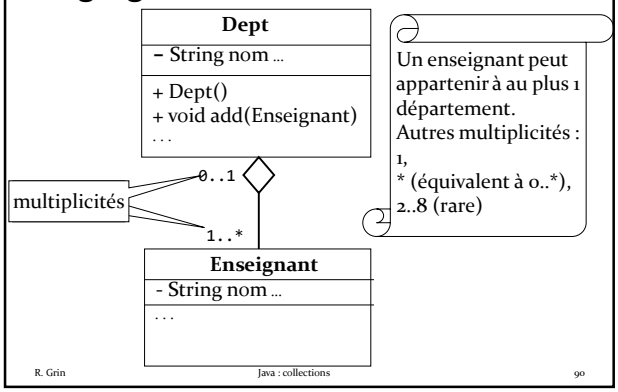
88

Exemple

```
listeEtudiants.sort(
  Comparator.comparingInt(Etudiant::getPromo)
  .reversed()
  .thenComparingDouble(Etudiant::getMoyenne);
```


89

L'agrégation en notation UML



90