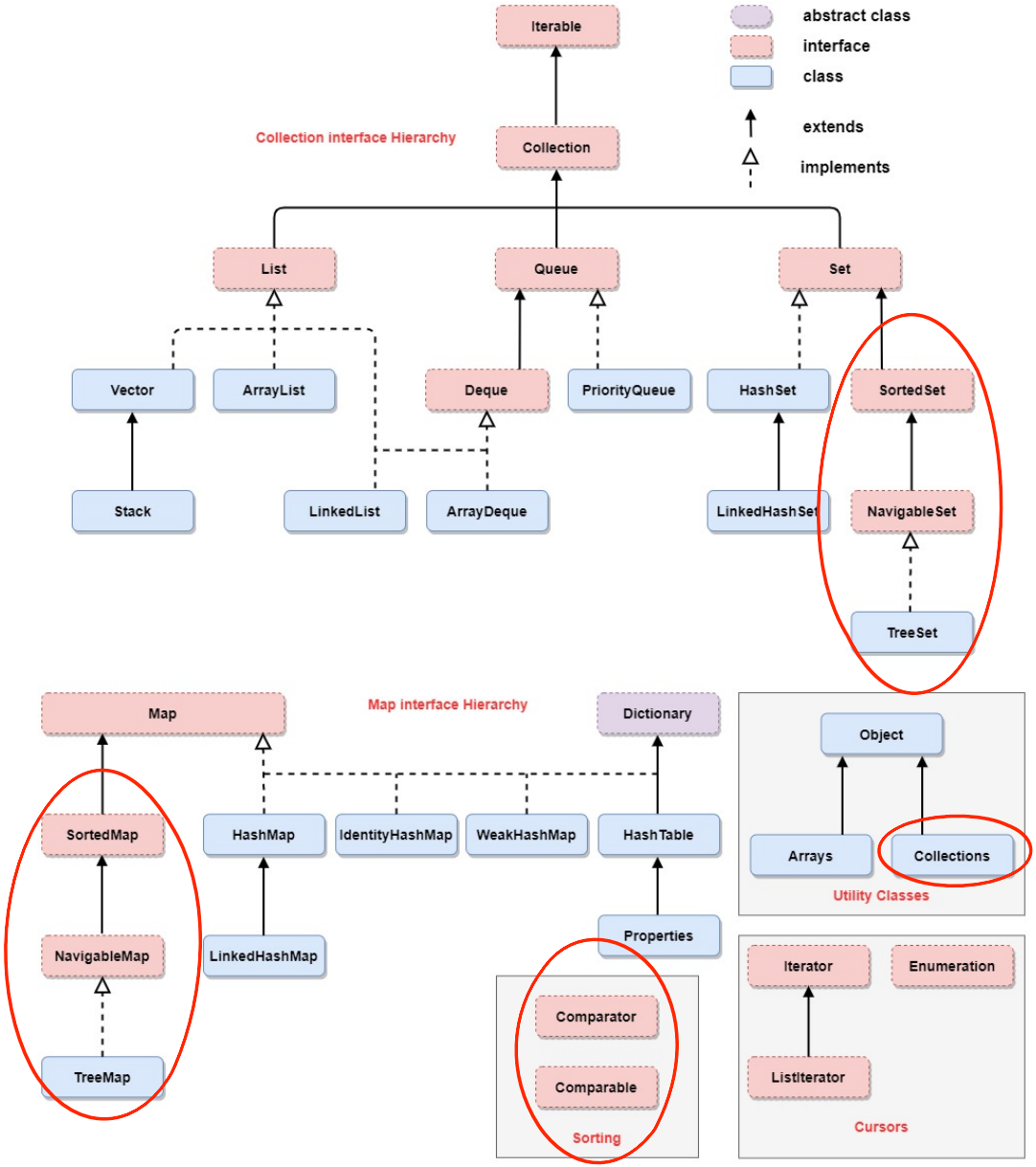
Ing39 – Collections Triées

Pierre Courtieu Virginia Aponte

Collections triees (https://vidvaan.com/java-collection-framework-overview/)



Dans ce cours

- ⋆ les collections de la bibliothèque triées « par construction »
- ⋆ les méthodes de comparaison à implanter
- ⋆ trier les listes avec `Collections.sort`

Conteneurs triés par construction

Idée : conteneur dont les éléments sont triés en permanence, selon un ordre établit à la création. Opérations+parcours préservent cet ordre.

Dans la bibliothèque :

- ⋆ `SortedSet<E>`, `NavigableSet<E>` ⇒ implantation `TreeSet<E>`
- ⋆ `SortedMap<K,V>`, `NavigableMap<K,V>` ⇒ implantation `TreeMap<K,V>`

Interfaces triés de la bibliothèque : `SortedSet<E>`

`SortedSet<E>` : collection de **`E`** sans doublons, triés en permanence.

Opérations « nouvelles » tirent parti du tri des éléments

- ⋆ **`first()`**, **`last()`** : suivant l'ordre
- ⋆ vues « ensemble trié » : **`headSet(E t)`**, **`tailSet(E t)`**, **`subSet(E inf, E sup)`**
- ⋆ `NavigableSet<E>` : étend **`SortedSet`** avec + opérations
- ⋆ Implantation : `TreeSet<E>`

Interfaces triés de la bibliothèque : `SortedMap<K,V>`

`SortedMap<K,V>` : dictionnaire associatif

- clés forment 1 ensemble trié **`SortedSet<K>`**
- **`firstKey()`** et **`lastKey()`**
- **`headMap(K k)`**, **`tailMap(K k)`** : vues **`Map`** pour jusqu'à/à partir clé k ...
- `NavigableMap<K,V>` : extension de **`SortedMap`** avec +opérations
- Implantation : `TreeMap<K,V>`

Établir un ordre pour conteneur<E> trié

Revient à implanter 1 méthode de comparaison entre éléments E (e_1, e_2), qui détermine l'ordre entreux eux . Son comportément :

- ⋆ si e_1 plus grand $e_2 \Rightarrow$ revoie entier positif
- ⋆ si e_1 plus petit $e_2 \Rightarrow$ revoie entier négatif
- ⋆ si e_1 égal $e_2 \Rightarrow$ renvoie 0

1 parmi 2 méthodes (et son interface) à implanter :

- ⋆ `e1.compareTo(E e2)` $\overset{\text{implanter}}{\Longrightarrow}$ interface Comparable<E>
- ⋆ `compare(E e1,E e2)` $\overset{\text{implanter}}{\Longrightarrow}$ interface Comparator<E>

En interne, opérations conteneur appelent 1 de ces méthodes pour maintenir ordre

- ⋆ Ex : `add(e)` compare avec éléments présents pour décider où insérer `e`

Implanter un ordre : `Comparable<E>` ou `Comparator<E>`

2 possibilités :

- ⋆ la classe **`E`** des éléments ⇒ implante **`Comparable<E>`**
- ⋆ OU, une *autre classe* **`Comp`** ⇒ implante **`Comparator<E>`**

Pourquoi 2 possibilités (et laquelle choisir) ?

- ⋆ implanter **`Comparable<E>`**, revient à ajouter **`compareTo`** dans E

  ⇒ à faire si l'ordre implanté est « l'ordre naturel pour E »

- ⋆ **`Comparator<E>`** sert à définir des « objets comparateurs » selon d'ordres ponctuels, différents de celui éventuellement dans E
  - ⋆ passés en paramètre à **`Collections.sort`**,
  - ⋆ ou, à un constructeur de collection trié

  ⇒ pour obtenir un tri différent

Implanter un ordre avec `Comparable<E>`

La classe **`E`** des éléments implante `Comparable<E>` :

- **`E`** doit incorporer la méthode **`compareTo(E)`**
  - ⇒ classe des éléments incorpore sa propre méthode de comparaison ou « ordre naturel »
    - Ex : ordre alphabétique des noms de **`Contact`**
- si **`E`** standard, implantée d'emblée (**`String`**, **`Integer`**, ...)
-  s'assurer que **`compareTo`** et **`equals`** sont cohérents :
  - ⇒ **`(e1.compareTo(e2) == 0) == (e1.equals(e2))`**
    - si e1 égal-par-CompareTo e2 ⇒ **`e1.equals(e2)`**
    - si **`e1.equals(e2)`** ⇒ e1 égal-par-CompareTo e2

Exemple Comparable<Compte>

L'ordre naturel des comptes ici est l'ordre croissant de leurs numéros.

```
public class Compte implements Comparable <Compte> {
   private int numero;
   private double solde;
   // constructeurs, méthodes d'instance ...

  @Override
   public int compareTo(Compte o) {
      return this.getNumero().compareTo(o.getNumero());
   }

   @Override
   public int hashCode() {...}
   @Override
   public boolean equals(Object obj) {...}
 }
```

Notez : **equals** et **hashCode** sont également rédéfinis.

On doit assurer la cohérence entre **compareTo** et **equals**.

Implanter un ordre avec un `Comparator<E>`

Une *autre classe* **`Comp`** implante `Comparator<E>`

- **`Comp`** doit définir la méthode **`compare(E,E)`**
- 1 instance de **`Comp`** est un *objet comparateur*
- Utilisation :
  - collection triée créee via constructor avec en paramètre un comparateur, préservera son ordre
    - ex : `new TreeSet<Article>(new ComparatorPrixArticle())`
  - opération **`sort`** dans **`List`** et **`Collections`** : paramètre comparateur
- **`Comparator`** différents $\Rightarrow$ conteneurs ou tri des listes avec ordres divers
  - Ex : ordre de prix, ou de quantité en stock, ou alphabétique pour Article

Exemple Comparator<Compte>

Ponctuellement, on peut vouloir trier les **Comptes** par odre de soldes.

```
public class OrdreParSolde implements Comparator<Compte>{
   public  int compare(Compte o1, Compte o2){
       if (o1.getSolde() < o2.getSolde()) {
               return -1;
       }else if (o1.getSolde() > o2.getSolde()){
               return 1;
       } else { return 0; }
   }
}
```

Suite exemple


```
Compte c1 = new Compte (1,2500);
Compte c2= new Compte (7, 200);
Compte c4 = new Compte (20, 100);
Compte c4Doublon = new Compte (20, 100);

// 4 comptes dans la liste, dont un doublon
ArrayList<Compte> lc = new ArrayList<Compte>();
lc.add(c1); lc.add(c2); lc.add(c4); lc.add(c4Doublon);

// un set trié par numéros de Compte
// car Compte implante Comparable<Compte>
NavigableSet<Compte> sn = new TreeSet<Compte>(lc);

// un set trié par solde de Compte
// car un objet Comparator est passé au constructeur
NavigableSet<Compte> ss = new TreeSet<Compte>(new OrdreParSolde());
ss.addAll(lc);

System.out.println("Ensemble trié par ordre de numéros ");
for (Compte c:sn) {
    System.out.println(c.toString());
}
System.out.println("Ensemble trié par ordre de soldes ");
for (Compte c:ss) {
     System.out.println(c.toString());
}
```

Affichages exemple


```
Ensemble trié par ordre de numéros

Numero: 1, solde: 2500.0
Numero: 7, solde: 200.0
Numero: 20, solde: 100.0

Ensemble trié par ordre de soldes

Numero: 20, solde: 100.0
Numero: 7, solde: 200.0
Numero: 1, solde: 2500.0
```

Notez que le doublon n'a pas été ajouté dans les ensembles. Nous avions redéfini `equals` et `hashCode`

`Collections.sort (List<E>)`

La classe `Collections` (notez le « s ») regroupe *méthodes statiques* utilitaires (recherche dichotomique, tri, etc.)

```
static void Collections.sort (List<E>)
```

Cette méthode a besoin de :

- une liste (`List<E>`) d'éléments à trier :
  - si notre collection n'est pas une liste (p.e. un `Set`, un `Map`) nous devrons en fabriquer une pour utiliser `Collections.sort` ;
- implanter 1 méthode de comparaison entre éléments :
  - (en plus de la liste à trier), passer en paramètre un objet `Comparator<E>`
  - OU, on passe seulement la liste à trier, où `E` implante `Comparable<E>`

FIN