جامعة هواري بومدين
للعلوم والتكنولوجيا
U S T H B

Chapitre 7

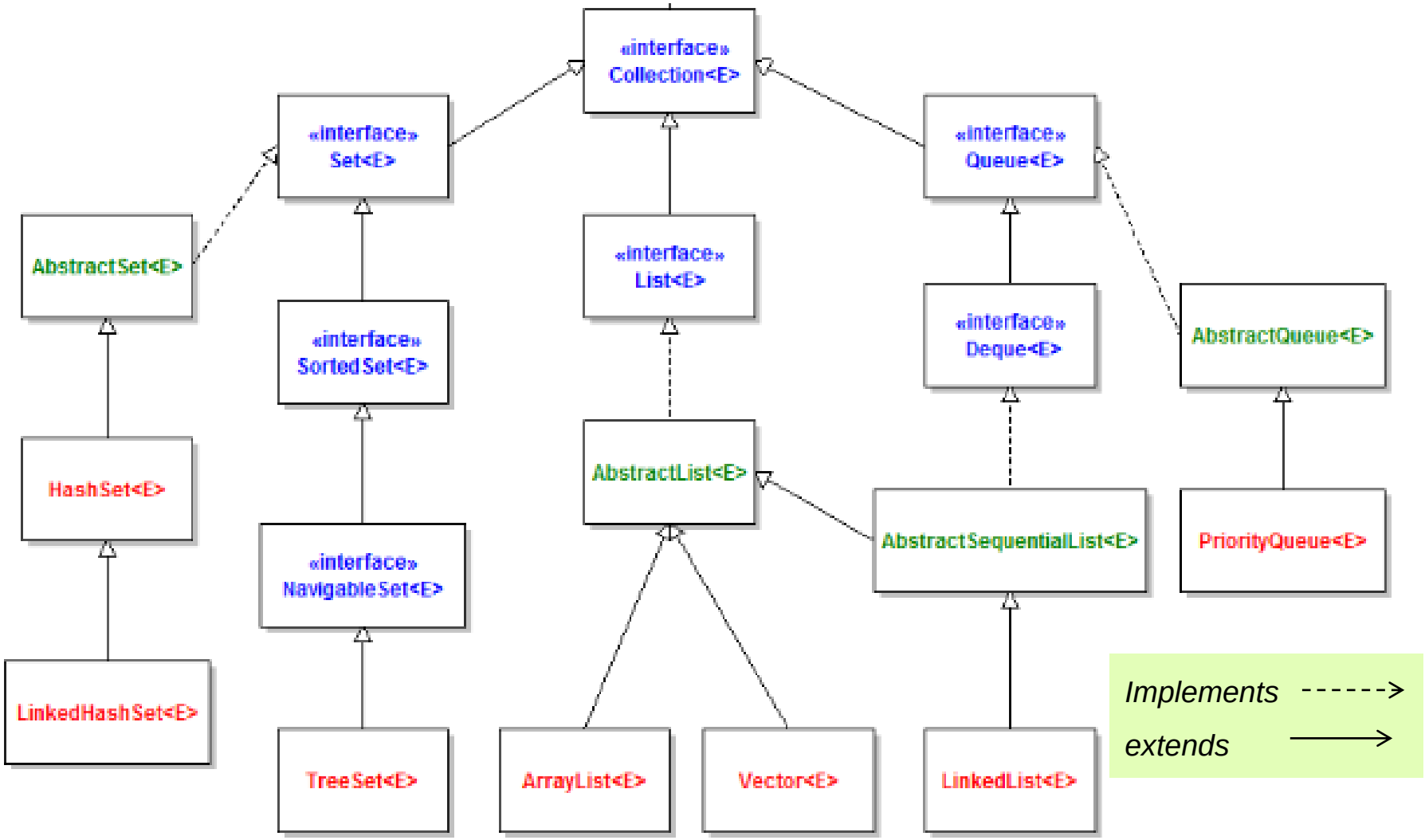
Les Collections en java

S. BOUKHEDOUMA

USTHB – FEI – département d'Informatique
Laboratoire des Systèmes Informatiques -LSI
sboukhedouma@usthb.dz

L'interface « Set »
Les ensembles

Le Framework des Collections – L'interface « Collection »



Les Collections – l'interface « Set »

L'interface « Set »

interface Set extends Collection {…}

Une collection de type Set est une collection où on ne considère pas **la notion d'ordre entre les éléments** (1er, 2ème, …)

Dans une collection de type Set, on n'accepte pas **les doublons (les répétitions d'éléments).**

La **notion d'index n'est pas** utilisée dans les méthodes de cette interface

Collections –l'interface « Set »

L'interface **« Set »:** **hérite** de toutes les signatures de méthodes de l'interface Collection et :

Méthode	Rôle
boolean add(E e)	Ajouter l'élément fourni en paramètre à la collection si celle-ci ne le contient pas déjà et renvoyer un booléen qui précise si la collection a été modifiée (l'implémentation de cette opération est optionnelle)
boolean addAll(Collection<? extends E> c)	Ajouter tous les éléments de la collection fournie en paramètre à la collection si celle-ci ne les contient pas déjà et renvoyer un booléen qui précise si la collection a été modifiée (l'implémentation de cette opération est optionnelle)
void clear()	Retirer tous les éléments de la collection (l'implémentation de cette opération est optionnelle)
boolean contains(Object o)	Renvoyer un booléen qui précise si la collection contient l'élément fourni en paramètre
boolean containsAll(Collection<? > c)	Renvoyer un booléen qui précise si tous les éléments de la collection fournie en paramètre sont contenus dans la collection
boolean equals(Object o)	Comparer l'égalité de la collection avec l'objet fourni en paramètre. L'égalité est vérifiée si l'objet est de type Set, que les deux collections ont le même nombre d'éléments et que chaque élément d'une collection est contenu dans l'autre

Collections –l'interface « Set »

boolean isEmpty()	Renvoyer un booléen qui précise si la collection est vide
Iterator<E> iterator()	Renvoyer un Iterator sur les éléments de la collection
boolean remove(Object o)	Retirer l'élément fourni en paramètre de la collection si celle-ci le contient et renvoyer un booléen qui précise si la collection a été modifiée (l'implémentation de cette opération est optionnelle)
boolean removeAll(Collection<?> c)	Retirer les éléments fournis en paramètres de la collection si celle-ci les contient et renvoyer un booléen qui précise si la collection a été modifiée. (l'implémentation de cette opération est optionnelle)
boolean retainAll(Collection<?> c)	Retirer tous les éléments de la collection qui ne sont pas dans la collection fournie en paramètre (l'implémentation de cette opération est optionnelle)
int size()	Renvoyer le nombre d'éléments de la collection. Si ce nombre dépasse Integer.MAX_VALUE alors la valeur retournée est MAX_VALUE
Object[] toArray()	Renvoyer un tableau des éléments de la collection
<T> T[] toArray(T[] a)	Renvoyer un tableau des éléments de la collection dont le type est celui fourni en paramètre

Les Collections – l'interface « Set »

Les implémentations de l'interface **« Set »** (*le lien implements*)

La classe HashSet

représente un ensemble d'éléments d'une taille variable et non trié

public abstract class AbstractSet implements Set {…
}
public class HashSet extends AbstractSet {…
//implémentation de toutes les méthodes de « Set »}

Constructeur	Rôle
HashSet()	Créer une nouvelle instance vide dont la HashMap interne utilisera une capacité initiale et un facteur de charge par défaut
HashSet(Collection<? extends E> c)	Créer une nouvelle instance contenant les éléments de la collection fournie en paramètre
HashSet(int initialCapacity)	Créer une nouvelle instance vide dont la HashMap interne utilisera la capacité initiale fournie en paramètre et un facteur de charge par défaut
HashSet(int initialCapacity, float loadFactor)	Créer une nouvelle instance vide dont la HashMap interne utilisera la capacité initiale et un facteur de charge par défaut

Les Collections – l'interface « Set »

Les implémentations de l'interface **« Set »** (*le lien implements*)

La classe HashSet

A la création d'une collection de type HashSet, il est possible de préciser deux paramètres:

- *la capacité initiale de la collection* : taille de la collection (nbre d'éléments)
- *le facteur de charge*: le pourcentage de remplissage de la collection pour décider d'augmenter sa taille (automatiquement).

Les valeurs par défaut sont:

la taille = **16**

facteur de charge = **0,75**

Les Collections – l'interface « Set »

La classe HashSet

Pour l'utiliser : import java.util.HashSet;

Exemple

```
Set <String>  S = new HashSet <String> ();
S.add (”aaaa” ); S.add (”xxxx" );
S.add (”zzzz” ); S.add (”cccc” );
Set <String>  T = new HashSet <String> (S);
                                    // on met dans T tous les éléments de S.
```

Equivalent à écrire:

```
Set <String>  T = new HashSet <String> (); // créer une collection vide
T.addAll(S);        // ajouter tous les éléments de S à T
System.out.println (T); // on peut utiliser une boucle for ou un « iterator »
```

On aura:

aaaa xxxx zzzz cccc

La classe HashSet

Exemple (suite)

```
HashSet <String>  S1 = new HashSet <String> ();
S1.add (”aaaa” ); S1.add (”zzzz" );
S.retainAll (S1);    // supprimer de S tous les éléments qui ne sont pas dans S1- intersection.
                                                    // affichage
Iterator <String> it = S1.iterator();
                While (it.hasNext())    System.out.print (it.next() + ”\t ”);
system.out.println();
Iterator <String> it2 = S.iterator();
                While (it2.hasNext())     System.out.print (it.next() + ”\t ”);
```

On aura:

aaaa zzzz (dans S1)

aaaa zzzz (dans S)

La classe HashSet

Exemple (suite)

On suppose dans **S** les éléments: ″aaaa″ ″xxxx″ ″zzzz″ ″cccc″

```
HashSet <String>  S1 = new HashSet <String> ();
S1.add (″aaaa″ ); S1.add (″zzzz" );
S.removeAll (S1);    // supprimer de S tous les éléments qui sont dans S1

                                        // affichage
Iterator <String> it = S.iterator();
            While (it.hasNext())     System.out.print (it.next() + ″\t ″);
```

On aura (dans S):

xxxx cccc

La classe HashSet

Exemple (suite)

```
Set <String>  S = new HashSet <String> ();
S.add (″aaaa″ ); S.add (″xxxx" ); S.add (″zzzz″ ); S.add (″cccc″ );

 // construire un tableau à partir de la collection
String   Tab[]  =   S.toArray();
                                        // affichage
for (String s: Tab)   System.out.print (s  +   ″t″  );
                                        ou bien
for (i = 0; i< Tab.length; i++)  System.out.print (Tab[i] +   ″t″ );
```

On aura (dans Tab):

aaaa xxxx zzzz cccc

La classe HashSet

Entre List et Set

Il est possible de récupérer les éléments d'une collection de type List (ArrayList ou LinkedList) dans une collection de type Set (HashSet, …) et vice versa.

Exemple

```
ArrayList <String>  A = new ArrayList <String> ();
A.add ("Hello" ); A.add ("good" );
A.add ("morning" ); A.add ("every body" );

Set <String>  S = new HashSet <String> (A);
                    // on met dans S tous les éléments de la liste A
```

Les Collections – l'interface « Set »

La classe HashSet

Exemple 2 (HashSet)

```
HashSet <Point>  C = new HashSet <Point>();
C.add (new Point(3,6) ); C.add (new Point(-2,2));
C.add (new Point (0, 5));  C.add (new Point (3,6));
C.add (new Point(1, -1));

Iterator <Point> it = C.iterator();
            While (it.hasNext())    System.out.print (it.next() + "\t ");
```

On aura:

(3,6) (-2,2) (0,5) (3,6) (1, -1) ou bien
(3,6) (-2,2) (0,5) (1, -1) // selon la redéfinition de la méthode equals

Les Collections – l'interface « Set »

La classe HashSet

Pour éviter les répétitions d'éléments dans un Set, la méthode add utilise la méthode equals sur les objets de la collection de type Set.

La méthode equals devra être redéfinie dans la classe d'objets pour vérifier si deux objets sont égaux.

Si l'élément à ajouter existe déjà dans la collection de type HashSet, il ne sera pas ajouté sans générer une erreur (une exception)

L'interface « SortedSet »
Les ensembles triés

Les Collections – l'interface « Set »

L'interface « Set »

interface Set extends Collection {…}

A partir de l'interface « Set », une autre interface est dérivée

L'interface **« SortedSet »** : permet de manipuler des ensembles d'éléments triés (garantit un stockage et parcours des éléments dans l'ordre croissant)

interface SorteSet extends Set {…}

Les Collections – l'interface « Set »

L'interface « SortedSet »

interface SortedSet extends Set {…}

Hérite de toutes les signatures de méthodes de l'interface « Set » et possède des signatures de méthodes supplémentaires:

Méthode	Rôle
E first()	Retourner le premier élément de la collection
E last()	Retourner le dernier élément de la collection
SortedSet headSet(E toElement)	Retourner un sous-ensemble des premiers éléments de la collection jusqu'à l'élément fourni en paramètre exclus
SortedSet tailSet(E fromElement)	Retourner un sous-ensemble contenant les derniers éléments de la collection à partir de celui fourni en paramètre inclus
SortedSet subSet(E fromElement, E toElement)	Retourner un sous-ensemble des éléments dont les bornes sont ceux fournis en paramètres. fromElement est inclus et toElement est exclus. Si les deux éléments fournis en paramètres sont les mêmes, la méthode renvoie une collection vide
Comparator< ? super E> comparator()	Renvoyer l'instance de type Comparator associée à la collection ou null s'il n'y en a pas

Les Collections – l'interface « SortedSet »

L'interface « SortedSet »

Comment définir l'ordre des éléments dans une collection de type SortedSet?

-la classe d'objets doit implémenter l'interface **Comparable** (la méthode compare To), définie dans java.lang

-Définir une classe qui implémente l'interface **Comparator** définie dans java.util

L'interface **Comparator** permet de définir des comparateurs sur plusieurs attributs de la même classe d'objets (nom, âge, …)

Les Collections – l'interface « SortedSet »

Les implémentations de l'interface **« SortedSet »** (*le lien implements*)

La classe TreeSet

représente un ensemble d'éléments d'une taille variable et **trié**

public class **TreeSet** implements SotedSet {…

//implémentation de toutes les méthodes de « SortedSet »}

Constructeur	Rôle
TreeSet()	Créer une instance vide dont l'ordre naturel de tri de ses éléments est utilisé
TreeSet(Collection<? extends E> c)	Créer une instance contenant les éléments de la collection fournie en paramètre dont l'ordre naturel de tri de ses éléments est utilisé
TreeSet(Comparator<? super E> comparator)	Créer une instance vide dont l'ordre utilisé est celui définit par l'instance de type Comparator fournie en paramètre
TreeSet(SortedSet<E> s)	Créer une instance contenant les éléments de la collection fournie en paramètre dont l'ordre est celui utilisé par la collection

les éléments d'un TreeSet sont organisés sous forme d'un arbre binaire ordonné (et équilibré) La recherche d'un élément se fait de manière dichotomique

La classe TreeSet

Exemple (treeSet)

```
Set <String>  S = new TreeSet <String> ();
S.add (″aaaa″ ); S.add (″xxxx" ); S.add (″zzzz″ ); S.add (″cccc″ );


                                        // affichage
Iterator <String> it = S.iterator();
            While (it.hasNext())     System.out.print (it.next() + ″\t ″);
```

On aura :

aaaa cccc xxxx zzzz

les éléments sont triés car la méthode compareTo est définie dans la classe String

La classe TreeSet

Exemple (TreeSet)

On suppose qu'on a une classe Personne avec deux attributs (nom, âge)
Pour créer un **TreeSet** d'objets Personne, il faut définir la *méthode de comparaison des éléments*

```
Public class Personne implements <Comparable>
{ //attributs
  //constructeur        //méthodes
   public int compareTo( Object obj)    //on compare les âges des personnes
        { Personne P = (Personne)obj;
            if  (this.age > P.age)  return 1;
            if (this.age < P.age) return -1;
            return 0; } }
```

La classe TreeSet

Exemple (TreeSet)

```
Set <Personne>  TS = new TreeSet <Personne> ();
TS.add (new Personne ("Ryma", 25);  TS.add (new Personne ("Khaled", 32);
TS.add (new Personne ("Nada", 14);  TS.add (new Personne ("Walid", 28); …
                                        // affichage
Iterator <String> it = S.iterator();
            While (it.hasNext())     System.out.println (it.next());
```

On aura :

```
Nada    14
Ryma    25
Walid   28
Khaled  32
```

// l'ordre défini par la méthode CompareTo

La classe TreeSet

Cohérence entre les méthodes « equals » et « compareTo »

Le mécanisme utilisé pour la comparaison lors de la définition de l'ordre (Comparable ou Comparator) doit être cohérent avec l'implémentation de la méthode equals() :

si element1.compareTo(element2) == 0
alors obligatoirement
element1.equals(element2) == true.

Les Collections – l'interface « NavigableSet»

L'interface NavigableSet hérite de SortedSet

Fourni des méthodes qui permettent de:

-Parcourir les éléments dans **l'ordre décroissant (et croissant)**
- Récupérer un sous-ensemble d'éléments plus grands (ou plus petits) qu'un élément donné en paramètre
- récupérer les éléments appartenant à un intervalle donné
-tronquer une partie du début ou la fin de l'ensemble des éléments
…

La classe qui implémente l'interface NavigableSet est toujours le TreeSet

Les Collections – l'interface « NavigableSet »

Autres méthodes

Méthode	Rôle
E ceiling(E e)	Retourner le plus petit élément qui soit plus grand ou égal à celui fourni en paramètre. Renvoie null si aucun élément n'est trouvé
Iterator<E> descendingIterator()	Retourner un Iterator qui permet le parcours dans un ordre descendant des éléments de la collection
NavigableSet<E> descendingSet()	Retourner un ensemble parcourable dans le sens inverse de l'ordre de la collection actuelle
E floor(E e)	Retourner le plus grand élément qui soit plus petit ou égal à celui fourni en paramètre. Renvoie null si aucun élément n'est trouvé
SortedSet<E> headSet(E toElement)	Retourner un ensemble qui contient les éléments de la collection qui sont strictement plus petits que celui fourni en paramètre
NavigableSet<E> headSet(E toElement, boolean inclusive)	Retourner un ensemble parcourable qui contient les éléments de la collection qui sont strictement plus petits (ou plus petits ou égaux si le paramètre inclusive vaut true) que celui fourni en paramètre
E higher(E e)	Retourner le plus petit élément qui soit strictement plus grand que celui fourni en paramètre. Renvoie null si aucun élément n'est trouvé
Iterator<E> iterator()	Retourner un Iterator qui permet le parcours des éléments dans l'ordre ascendant
E lower(E e)	Retourner le plus grand élément qui soit strictement plus petit que celui fourni en paramètre. Renvoie null si aucun élément n'est trouvé

Autres méthodes

E pollFirst()	Retourner le premier élément et le retirer de la collection. Renvoie null si la collection est vide
E pollLast()	Retourner le dernier élément et le retirer de la collection. Renvoie null si la collection est vide
NavigableSet<E> subSet(E fromElement, boolean fromInclusive, E toElement, boolean toInclusive)	Retourner un sous-ensemble parcourable qui contient les éléments compris entre les deux éléments fournis en paramètres. Un booléen permet de préciser si chaque borne doit être incluse ou non.
SortedSet<E> subSet(E fromElement, E toElement)	Retourner un sous-ensemble qui contient les éléments compris entre le premier fourni en paramètre inclus et le second exclu
SortedSet<E> tailSet(E fromElement)	Retourner un sous-ensemble des éléments qui sont plus grands ou égaux à celui fourni en paramètre
NavigableSet<E> tailSet(E fromElement, boolean inclusive)	Retourner un ensemble parcourable qui contient les éléments de la collection qui sont strictement plus grands (ou plus grands ou égaux si le paramètre inclusive vaut true) que celui fourni en paramètre

La classe TreeSet

Exemple (TreeSet)

```
NavigableSet  <String> NS = new TreeSet <String>();

for (int i = 1; i < 10; i++) { NS.add("" + i*10); }   // chaines numériques

System.out.println(NS); // tous les éléments de la collection

System.out.println("ceiling(45)=" + NS.ceiling("45"));
System.out.println("floor(50)" + NS.floor("50"));
System.out.println("higher(70)=" + NS.higher("70"));
 System.out.println("lower(70)=" + NS.lower("70"));
…
```

On aura :

[10 20 30 40 50 60 70 80 90]
ceiling(45) = 50
floor(50) = 50
Higher (70) = 80
lower (70) = 60

Les accès concurrents (cas des multi-thread)

Notion de Thread : processus léger , un thread est une unité d'exécution (ensemble d'instructions) faisant partie d'un programme.

Cette unité fonctionne de façon autonome et parallèlement à d'autres threads.

Le principal avantage des threads est de pouvoir répartir différents traitements d'un même programme en plusieurs unités distinctes pour permettre leurs exécutions "simultanées".

Les accès concurrents (cas de multi-threads)

Problème

Plusieurs threads (processus) peuvent partager les objets du programme comme les objets de type Collection

Conséquence

Accès simultanée à une même collection (ArrayList, Hashset, Treeset, …)
Pour éviter les incohérences lors de modification de la collection, on utilise des classes **thread-safe**

Solution

Thread-safe : permettent de contrôler l'accès pour permettre une seule modification à la fois.

La classe TreeSet

Accès concurrents (multi-threads)

La classe TreeSet n'est pas thread-safe : comme aucune de **ses méthodes n'est synchronized**, un seul thread doit pouvoir modifier le contenu de la collection.

Si plusieurs threads doivent pouvoir modifier la collection, il faut invoquer la méthode synchronizedSortedSet() de la classe Collections qui va créer un wrapper dont les méthodes sont synchronized.

```
SortedSet set = Collections.synchronizedSortedSet(new TreeSet());
```

La classe TreeSet

Accès concurrents (multi-threads)

Certaines classes on été définies pour être thread-safe

En effet, elles permettent à plusieurs programmes (threads) de modifier la même collection.
Un mécanisme de *contrôle d'accès* est implémenté et utilisé pour garder la *cohérence* des données dans la collections)

La classe : CopyOnWriteArraySet

La classe : ConcurrentSkipListMap

L'interface « Map »
Les tables de hachage

Le Framework des Collections