Les collections génériques en Java

Cours de Programmation Orientée Objet Avancée

ISET Bizerte

CHALOUAH Anissa

Introduction au framework Collections

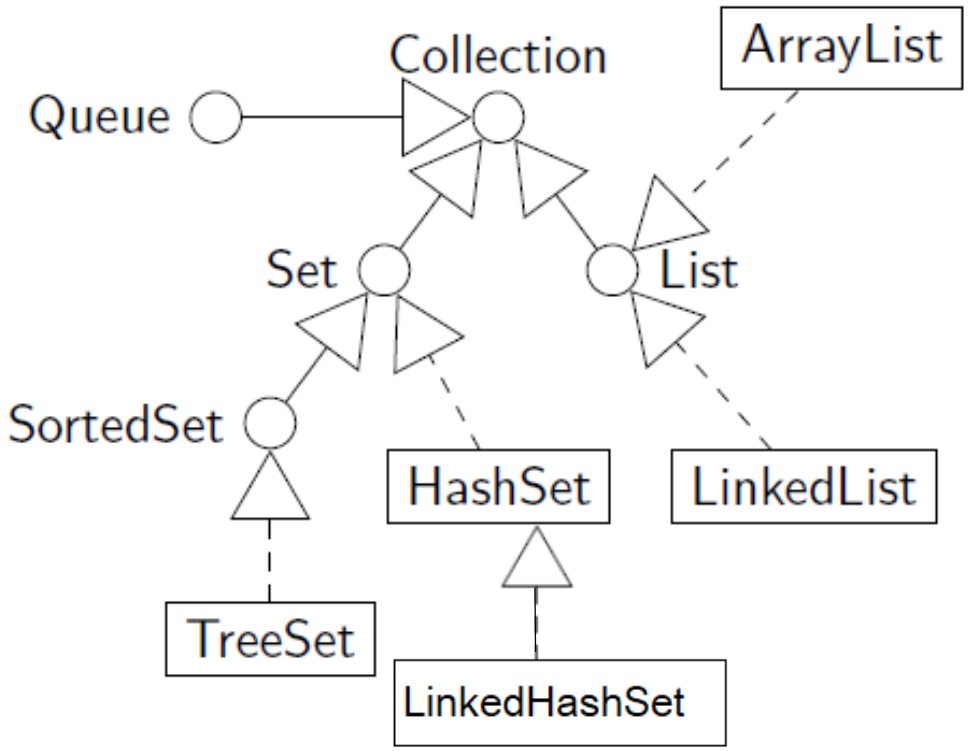
Introduction aux collections

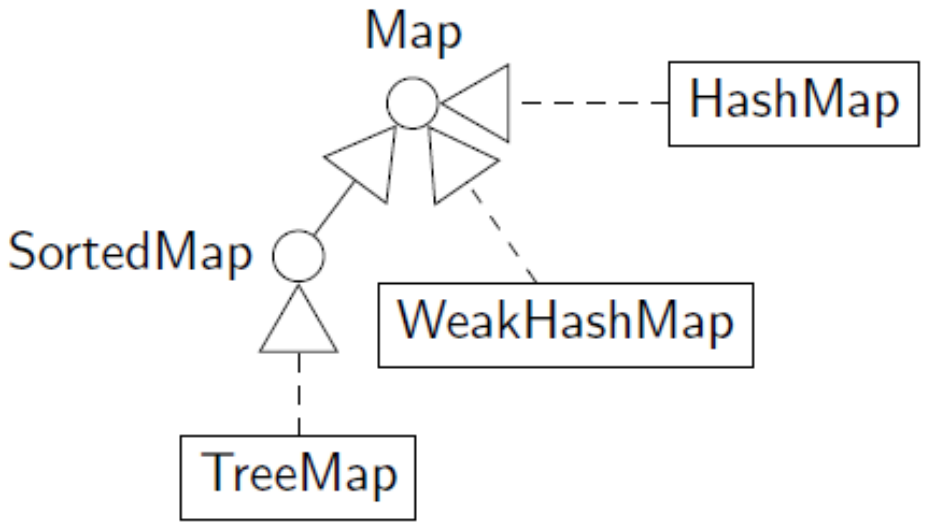
- les **collections** sont des **objets** qui permettent de **stocker** et de **manipuler** des objets autrement qu'avec un tableau conventionnel.
- Elles permettent de stocker des objets de différentes manières :
  - sous la forme d'une pile ;
  - comme une liste chaînée ;
  - sous la forme d'une structure clé-valeur ;
  - ...
- Ces formes de stockage seraient impossibles à faire avec un tableau. Elles sont généralement utilisées dans ce qu'on appelle un **framework**.

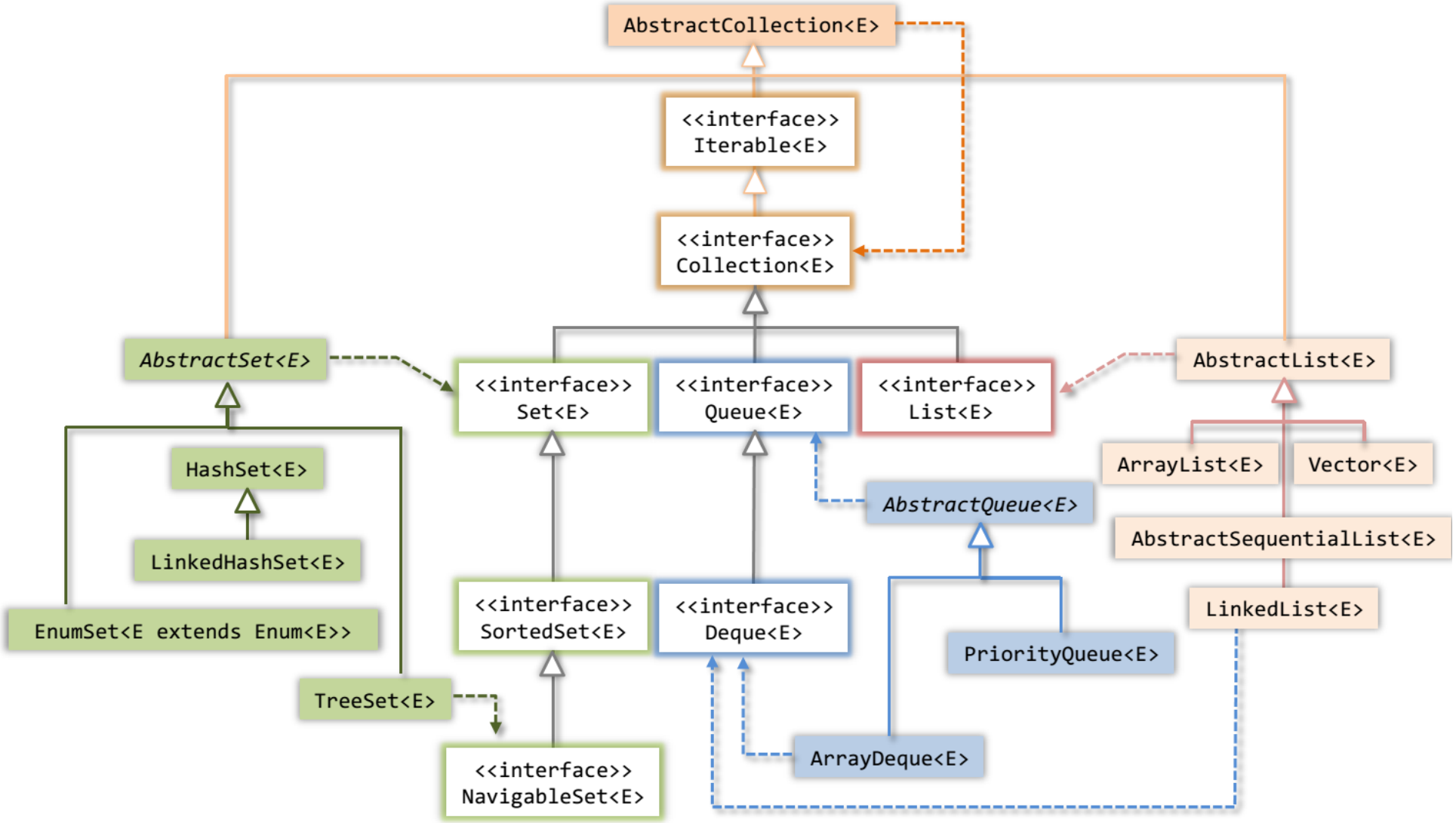
Introduction aux collections

- Les collections Java forment un **framework** qui permet de gérer des structures d'objets.
- Ce framework offre une architecture unifiée pour représenter et manipuler les collections(recherche, tri, etc.).
- Ce framework est constitué d'un ensemble **d'interfaces** dont les différentes fonctionnalités sont implémentées par des **classes concrètes**.

Les collections : classes et interfaces de base






AbstractCollection<E>
<<interface>>
Iterable<E>
<<interface>>
Collection<E>
AbstractSet<E>
<<interface>>
Set<E>
<<interface>>
Queue<E>
<<interface>>
List<E>
AbstractList<E>
HashSet<E>
LinkedHashSet<E>
EnumSet<E extends Enum<E>>
TreeSet<E>
AbstractQueue<E>
ArrayList<E>
Vector<E>
AbstractSequentialList<E>
LinkedList<E>
<<interface>>
SortedSet<E>
<<interface>>
Deque<E>
PriorityQueue<E>
ArrayDeque<E>
<<interface>>
NavigableSet<E>

Interfaces globales

- Ce sont les interfaces dont toutes les autres se servent afin d'avoir un comportement global commun.
  - **Collection** : interface qui est implémentée par la plupart des objets qui gèrent des collections
  - **Map** : interface qui définit des méthodes pour des objets qui gèrent des **collections sous la forme clé/valeur**

Les collections : interfaces de premier niveau

- **Set** : interface pour des objets qui **n'autorisent pas de doublons** dans l'ensemble. Il est indiqué pour gérer des éléments uniques : un calendrier, une liste de cartes à jouer etc.
- **List** : interface pour des objets qui **autorisent des doublons** et **un accès direct** à un élément.
- **Queue** : ce type de collections peut s'apparenter à une file d'attente. Ce sera à vous de gérer la façon d'ordonner les éléments qu'elles contiennent
- **SortedMap** : interface qui étend l'interface Map et permet d'**ordonner l'ensemble** par ordre croissant, très utile pour des listes de numéros de téléphone, de dictionnaire etc.

Les collections : interfaces de second niveau

Ce deuxième niveau d'interfaces, présentées ci-dessous, va vous permettre de rajouter des fonctionnalités et/ou contraintes supplémentaires à vos collections :

- **SortedSet** : interface qui étend l'interface Set et permet **d'ordonner l'ensemble par ordre croissant**
- **Deque** : permet d'insérer et d'enlever des éléments aux deux bouts de la collections, un peu comme une pile de carte.

Les collections : classes d'implémentation

Le framework propose plusieurs objets qui implémentent ces interfaces et qui peuvent être directement utilisés :

- **HashSet** : Hashtable qui implémente l'interface Set
- **TreeSet** : arbre qui implémente l'interface SortedSet
- **ArrayList** : tableau dynamique qui implémente l'interface List
- **LinkedList** : liste doublement chaînée (parcours de la liste dans les deux sens) qui implémente l'interface List
- **HashMap** : Hashtable qui implémente l'interface Map
- **TreeMap** : arbre qui implémente l'interface SortedMap

Parcours de collection

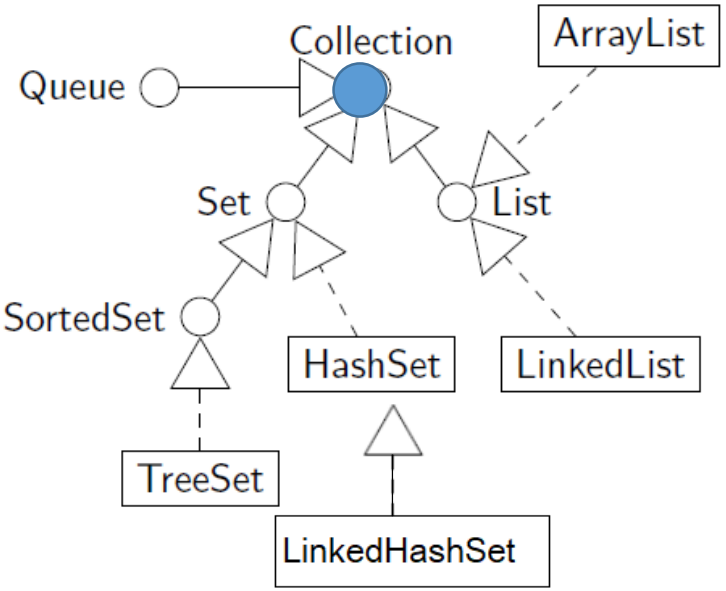
Le framework définit aussi des **interfaces** pour faciliter le parcours des collections et leur tri :

- **Iterator** : interface pour le parcours des collections
- **ListIterator** : interface pour le parcours des listes dans les deux sens et pour modifier les éléments lors de ce parcours
- **Comparable** : interface pour définir un ordre de tri naturel pour un objet
- **Comparator** : interface pour définir un ordre de tri quelconque

Interface Collection<E>

Interface Collection<E>

- Collection : L'interface **Collection<E>** correspond à un objet qui contient un groupe d'objets de type **E.**
- Elle représente un minimum commun pour les objets qui gèrent des collections :
  - ajout d'éléments, suppression d'éléments, vérification de la présence d'un objet dans la collection, parcours de la collection et quelques opérations diverses sur la totalité de la collection.
- Chaque implémentation de l'interface Collection devrait fournir au moins deux constructeurs :
  - un constructeur par défaut (sans argument)
  - un constructeur qui attend en paramètre un objet de type collection qui va créer une collection contenant les éléments de la collection fournie en paramètre



Interface Collection<E> : Méthodes

Méthode	Rôle
boolean add(E e)	Ajouter un élément à la collection (optionnelle)
boolean addAll(Collection<? extends E> c)	Ajouter tous les éléments de la collection fournie en paramètre dans la collection (optionnelle)
void clear()	Supprimer tous les éléments de la collection (optionnelle)
boolean contains(Object o)	Retourner un booléen qui précise si l'élément est présent dans la collection
boolean containsAll(Collection<?> c)	Retourner un booléen qui précise si tous les éléments fournis en paramètres sont présents dans la collection
boolean isEmpty()	Retourner un booléen qui précise si la collection est vide
Iterator<E> iterator()	Retourner un Iterator qui permet le parcours des éléments de la collection
boolean remove(Object o)	Supprimer un élément de la collection s'il est présent (optionnelle)
boolean removeAll(Collection<?> c)	Supprimer tous les éléments fournis en paramètres de la collection s'ils sont présents (optionnelle)
int size()	Retourner le nombre d'éléments contenus dans la collection
Object[] toArray()	Retourner un tableau contenant tous les éléments de la collection
<T> T[] toArray(T[] a)	Retourner un tableau typé de tous les éléments de la collection

L'interface Iterator

- Cette interface définit des méthodes pour des objets capables de parcourir les données d'une collection.

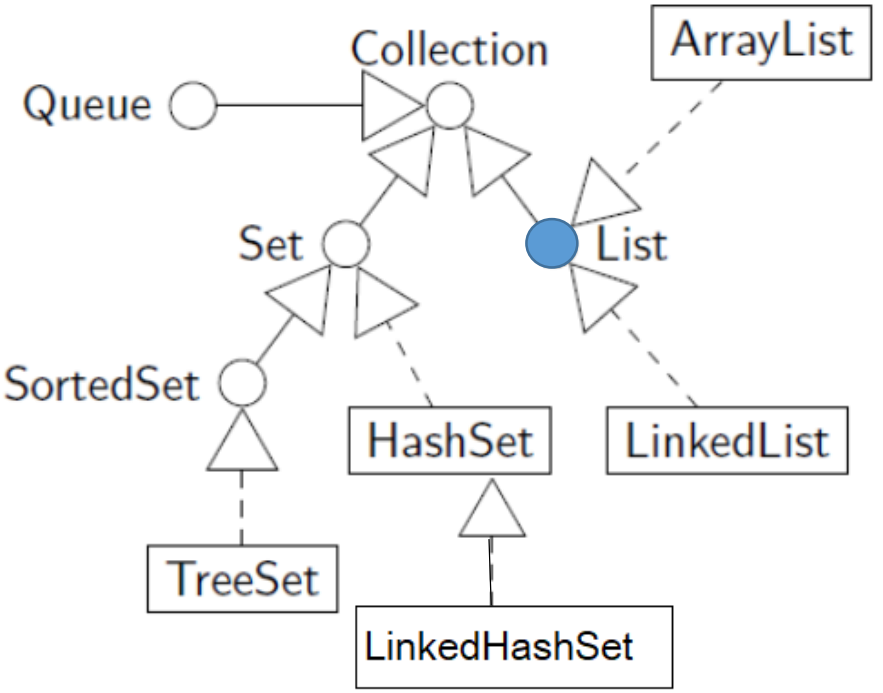
Méthode	Rôle
boolean hasNext()	Indiquer s'il reste au moins un élément à parcourir dans la collection
Object next()	Renvoyer le prochain élément dans la collection
void remove()	Supprimer le dernier élément parcouru


```
Iterator iterator = collection.iterator();
while (iterator.hasNext()) {
        System.out.println("objet = "+iterator.next());
}
```

Les collections de Type List : Les listes

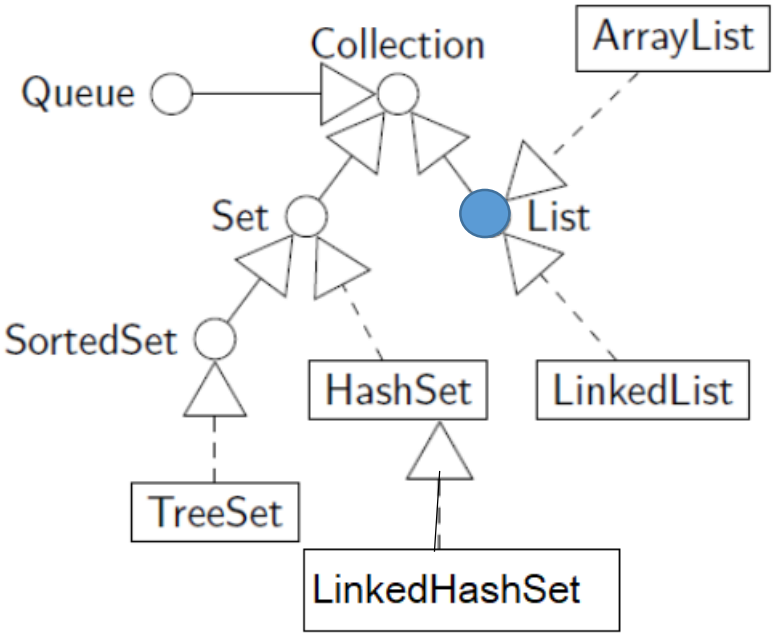
L'interface List (1)

- Cette interface, ajoutée à Java 1.2, étend l'interface **Collection**.
- Pour simplifier, les objets de type List sont des tableaux extensibles à volonté.
  - Pas de crainte de débordement.
- vous pouvez récupérer les éléments de la liste via leurs indices.



L'interface List(2)

- **List** est une collection **ordonnée** d'objets qui permet :
  - de contenir des doublons
  - d'interagir avec un élément de la collection en utilisant sa position (indice)
  - d'insérer des éléments null
- Pour les listes, une interface particulière est définie pour permettre le parcours dans les deux sens de la liste et réaliser des mises à jour : l'interface **ListIterator**

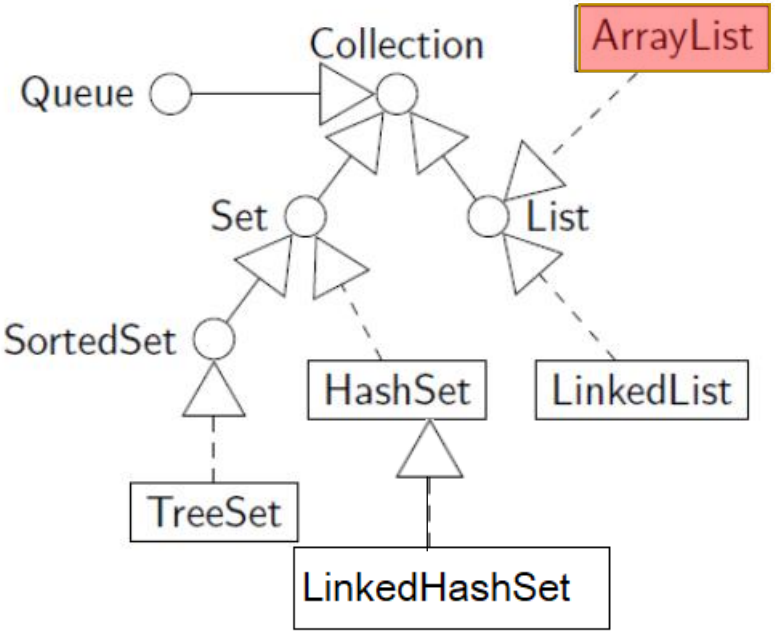


L'interface List : Méthodes

Méthode	Rôle
void add(int index, E e)	Ajouter un élément à la position fournie en paramètre
E get(int index)	Retourner l'élément à la position fournie en paramètre
int indexOf(Object o)	Retourner la première position dans la liste du premier élément fourni en paramètre. Elle renvoie -1 si l'élément n'est pas trouvé
int lastIndexOf(Object o)	Retourner la dernière position dans la liste du premier élément fourni en paramètre. Elle renvoie -1 si l'élément n'est pas trouvé
ListIterator<E> listIterator()	Renvoyer un Iterator positionné sur le premier élément de la liste
ListIterator<E> listIterator(int indx)	Renvoyer un Iterator positionné sur l'élément dont l'index est fourni en paramètre
E remove(int index)	Supprimer l'élément à la position fournie en paramètre
E set(int index, E e)	Remplacer l'élément à la position fournie en paramètre
List<E> subList(int fromIndex, int toIndex)	Obtenir une liste partielle de la collection contenant les éléments compris entre les index fromIndex inclus et toIndex exclus fournis en paramètres

La classe ArrayList<E>

- **ArrayList<E>** est un tableau **dynamique :** la taille (nombres d'éléments) du tableau n'est pas fixe et peut varier en cours d'exécution
- Une instance de la classe **ArrayList<E>** est une sorte de tableau qui peut contenir un nombre quelconque d'instances d'une classe E
- Comme pour un tableau l'accès à ses éléments est direct. Les emplacements sont indexés par des nombres entiers **(à partir de 0)**



La classe ArrayList<E> : Constructeurs

Constructeur	Rôle
ArrayList()	Créer une instance vide de la collection avec une capacité initiale de 10
ArrayList(Collection<? extends E> c)	Créer une instance contenant les éléments de la collection fournie en paramètre dans l'ordre obtenu en utilisant son iterator
ArrayList(int initialCapacity)	Créer une instance vide de la collection avec la capacité initiale fournie en paramètre

```
ArrayList <Personne> l1 = new ArrayList<>();
ArrayList <Personne> l2 = new ArrayList<>(5);
ArrayList <Personne> l3=new ArrayList<>(l1);
```

La classe ArrayList<E> : Méthodes

Méthode	Rôle
boolean add(E elt)	Ajouter un élément à la fin du tableau
void add(int indice, E elt)	Ajouter un élément à l'indice passé en paramètre
E get(int indice)	Renvoyer l'élément du tableau dont la position est précisée
int indexOf(Object obj)	Renvoyer la position de la première occurrence de l'élément fourni en paramètre
Iterator<E> iterator()	Renvoyer un itérateur sur le tableau
E remove(int indice)	Supprimer dans le tableau l'élément fourni en paramètre
E set(int indice, E elt)	Remplacer l'élément à la position indiquée par celui fourni en paramètre
int size()	Renvoyer le nombre d'éléments du tableau

La classe ArrayList<E> : Exemple(1)

```
ArrayList <Personne> l1 = new ArrayList<>();

l1.add(new Personne(1234,"amri","Salah"));
l1.add(new Personne(5678,"mekni","mohamed"));

System.out.println("\n **** Parcours liste 1 avec une bouble for ****");
for(int i=0;i<l1.size();i++)
        System.out.println(l1.get(i));
```

```
 **** Parcours liste 1 avec une bouble for ****
Personne [cin=1234, nom=amri, prenom=Salah]
Personne [cin=5678, nom=mekni, prenom=mohamed]
```

La classe ArrayList<E> : Exemple (2)

```
ArrayList <Etudiant> l2 = new ArrayList<>(5);
l2.add(new Etudiant(5555,"Toki","Sarra","DSI",2));
l2.add(new Etudiant(3333,"Jmili","Houda","RSI",3));
System.out.println("\n **** Parcours liste 2 avec bouble for each****");
for (Personne p : l2)
        System.out.println(p);
```

```
**** Parcours liste 2 avec une bouble for each****
Etudiant [filiere=DSI, niveau=2, toString()=Personne [cin=5555, nom=Toki, prenom=Sarra]]
Etudiant [filiere=RSI, niveau=3, toString()=Personne [cin=3333, nom=Jmili, prenom=Houda]]
```

La classe ArrayList<E> : Exemple (3)

```
ArrayList <Personne> l3=new ArrayList<>(l1);
System.out.println("Taille liste 3 = "+l3.size());
System.out.println("\n **** Parcours liste 3 avec un itérateur ****");
Iterator<Personne> i3=l3.iterator();
while (i3.hasNext()){
       System.out.println(i3.next());
```

```
Taille liste 3 = 2

 **** Parcours de la liste 3 avec un itérateur ****
Personne [cin=1234, nom=amri, prenom=Salah]
Personne [cin=5678, nom=mekni, prenom=mohamed]
```

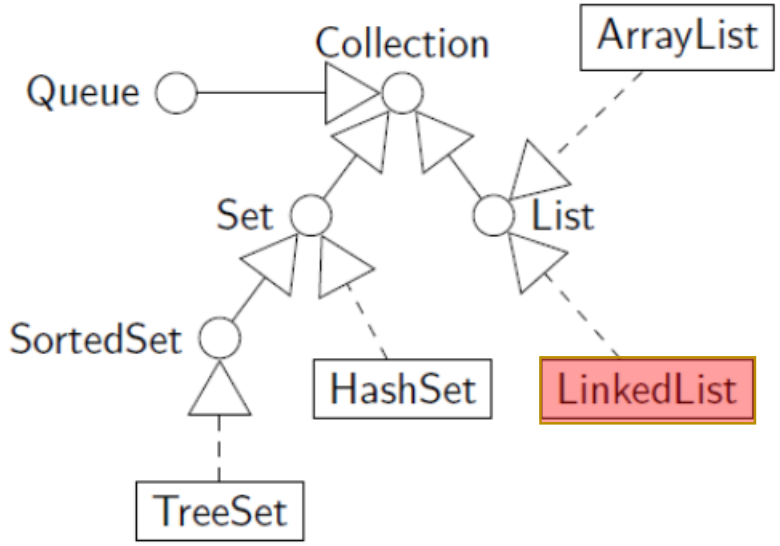
```java
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.ListIterator;
public class ArrayListExemple2 {
   public static void main(String[] args) {
      List<String> list = new
ArrayList<String>();
      list.add("a");
      list.add("b");
      list.add("c");
      list.add("d");
      list.add("e");
      list.add("f");

      //On met la liste dans le désordre
      Collections.shuffle(list);
      System.out.println(list);

      //On la remet dans l'ordre
      Collections.sort(list);
      System.out.println(list);

      Collections.rotate(list, -1);
      System.out.println(list);
```

```java
//On récupère une sous-liste
      List<String> sub = list.subList(2, 5);
      System.out.println(sub);
      Collections.reverse(sub);
      System.out.println(sub);

      //On récupère un ListIterator
      ListIterator<String> it = list.listIterator();
      while(it.hasNext()){
         String str = it.next();
         if(str.equals("d"))
            it.set("z");
      }
      while(it.hasPrevious())
         System.out.print(it.previous());

   }
}
```

```
[b, d, f, e, a, c]
[a, b, c, d, e, f]
[b, c, d, e, f, a]
[d, e, f]
[f, e, d]
azefcb
```

La classe LinkedList <E>

- La classe **LinkedList** est une implémentation d'une **liste doublement chaînée** dans laquelle les éléments de la collection sont reliés par des **pointeurs**.
- La liste peut être parcourue par un itérateur bidirectionnel **ListIterator**
- la classe **LinkedList** se prête bien à l'implémentation des collections **ordonnées**, c'est-à-dire
  - pile
  - queue (file d'attente)
  - séquence



La classe LinkedList <E> : Constructeurs

Constructeur	Rôle
LinkedList()	Créer une nouvelle instance vide
LinkedList(Collection<? extends E> c)	Créer une nouvelle instance contenant les éléments de la collection fournie en paramètre triés dans l'ordre obtenu par son Iterator

```
LinkedList <String> ll = new LinkedList<>();
ll.add("element 1");
ll.add("element 2");
ll.add("element 3");
Iterator<String> iterator = ll.iterator();
while (iterator.hasNext()) {
        System.out.println("objet = "+iterator.next());
}
```

```
objet = element 1
objet = element 2
objet = element 3
```

La classe LinkedList <E> : Méthodes

Méthode	Rôle
void addFirst(Object)	Insèrer l'objet au début de la liste
void addLast(Object)	Insèrer l'objet à la fin de la liste
Object getFirst()	Renvoyer le premier élément de la liste
Object getLast()	Renvoyer le dernier élément de la liste
Object removeFirst()	Supprimer le premier élément de la liste et renvoie l'élément qui est devenu le premier
Object removeLast()	Supprimer le dernier élément de la liste et renvoie l'élément qui est devenu le dernier

La classe LinkedList <E> : itérateur ListIterator

- L'interface **ListIterator** définit des fonctionnalités d'un **Iterator** permettant aussi le **parcours en sens inverse** de la collection, l'ajout d'un élément ou la modification du courant.

La classe LinkedList <E> : itérateur ListIterator

- En plus des méthodes définies dans l'interface Iterator dont elle hérite, l'interface ListIterator définit plusieurs méthodes :

Méthode	Rôle
void add(E e)	Ajouter un élément dans la collection
boolean hasPrevious()	Retourner true si l'élément courant possède un élément précédent
int nextIndex()	Retourner l'index de l'élément qui serait retourné en invoquant la méthode next()
E previous()	Retourner l'élément précédent dans la liste
int previousIndex()	Retourner l'index de l'élément qui serait retourné en invoquant la méthode previous()
void set(E e)	Remplacer l'élément courant par celui fourni en paramètre

La classe LinkedList <E> : Exemple

```
LinkedList<Integer> list=new LinkedList<Integer>();

  list.add(1);
      list.add(2);
      list.add(3);
      list.add(9);
      list.add(5);

      Iterator<Integer> iterator = list.iterator();
      while (iterator.hasNext()) {
      System.out.print(" "+iterator.next());
      }

     System.out.println();
```

```
 ListIterator<Integer> li=list.listIterator();

      while (li.hasNext()){
if (li.next()==9)li.set(4);}
      list.addFirst(0);
      list.addLast(10);

      for(int i=0;i<list.size();i++)
System.out.print(" "+list.get(i));
```

```
1 2 3 9 5
0 1 2 3 4 5 10
```

Les collections de type Set : les ensembles

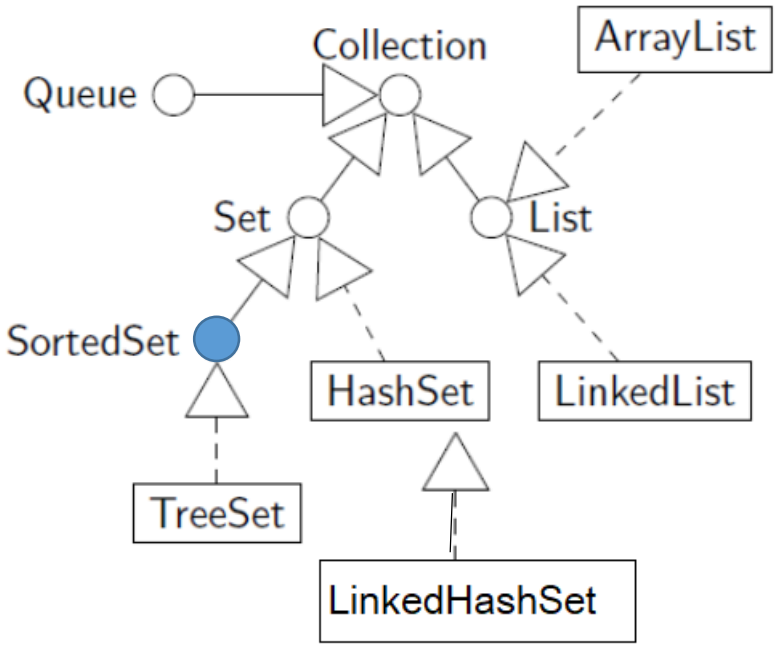
Les ensembles : L'interface Set

- Un ensemble est une collection **non ordonnée** d'éléments de type *E*, **aucun élément ne peut apparaître plus d'une fois dans un ensemble**
- **Problème**: comme deux objets distincts ont des références différentes, on ne pourra jamais avoir deux objets égaux même si toutes leurs valeurs sont identiques -> Il faudra définir un comparateur qui sera capable de tester l'égalité de deux objets (***equals*** et ***compareTo***)
- L'utilisateur devra définir, pour l'utilisation d'un
  - **HashSet** -> les méthodes ***hashCode*** et ***equals*** dans la classe des éléments *E*
  - **TreeSet** -> la méthode ***compareTo*** dans la classe *E*

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 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
int hashCode()	Retourner la valeur de hachage de la collection
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)
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

L'interface SortedSet

- L'interface SortedSet, ajoutée à Java 1.2, définit les fonctionnalités pour une collection de type Set qui **garantit l'ordre ascendant** du parcours de ses éléments.

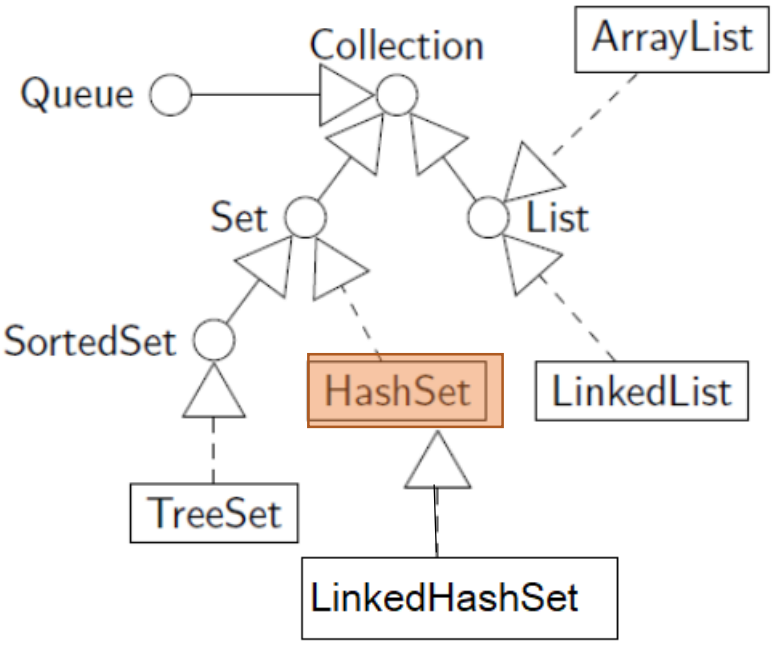


L'interface SortedSet

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

La classe HashSet

- La classe HashSet, ajoutée à Java 1.2, est une implémentation simple de l'interface Set .
- La classe HashSet présente plusieurs caractéristiques :
  - elle ne propose **aucune garantie sur l'ordre de parcours** lors de l'itération sur les éléments qu'elle contient
  - elle **ne** permet **pas d'ajouter des doublons** mais elle permet l'ajout d'un élément null



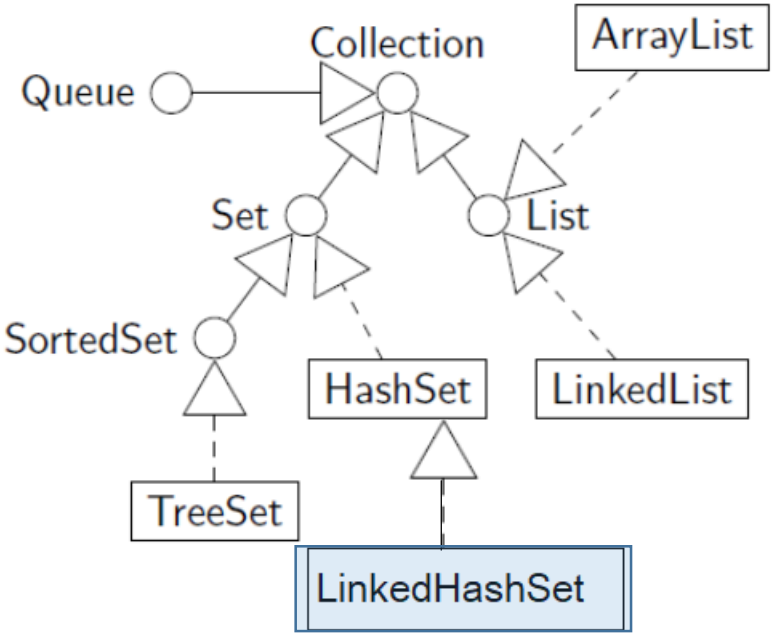
La classe HashSet : Exemple

```
import java.util.HashSet;
import java.util.Iterator;

public class HashSetExemple {
   public static void main(String[] args) {
   HashSet set = new HashSet();
    set.add("CCCCC");
    set.add("BBBBB");
    set.add("DDDDD");
    set.add("BBBBB");
    set.add("AAAAA");

    Iterator iterator = set.iterator();
    while (iterator.hasNext()) {
      System.out.println(iterator.next());
    }
   }
}
```

```
BBBBB
AAAAA
DDDDD
CCCCC
```

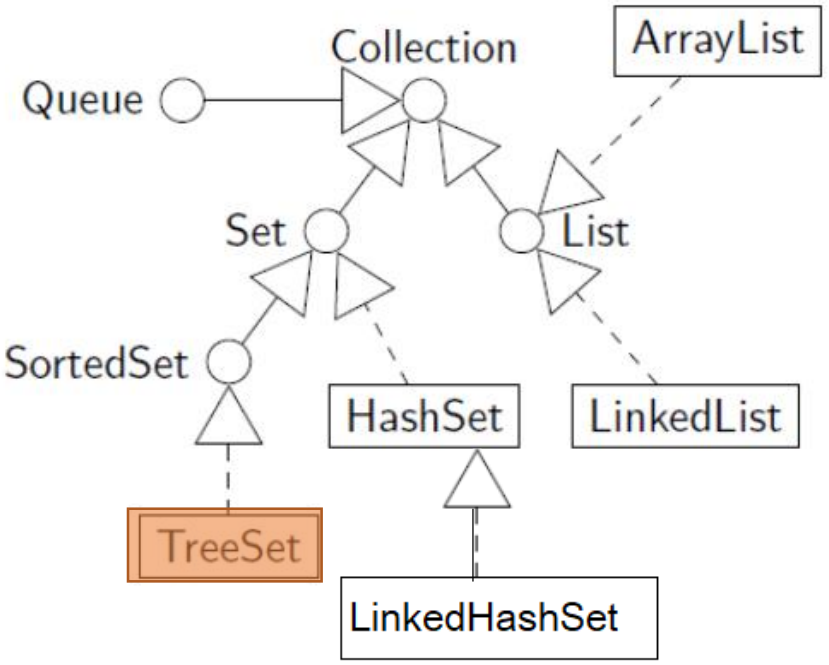
LinkedHashSet<E>

- **Set** qui maintient les clés dans **l'ordre d'insertion**
- Les performances sont légèrement moins bonnes que pour un **HashSet**



La classe TreeSet

- La classe TreeSet, ajoutée à Java 1.2, stocke ses éléments de manière ordonnée en les comparant entre-eux.
- Cette classe permet d'insérer des éléments dans n'importe quel ordre et de restituer ces éléments dans un ordre précis lors de son parcours.
- Une collection de type TreeSet ne peut pas contenir de doublons.
- TreeSet garantit que les éléments sont rangés dans **leur ordre naturel** (interface Comparable) ou l'ordre d'un Comparator.



La classe TreeSet : Exemple

```
import java.util.Iterator;
import java.util.TreeSet;

public class TreeSetExemple {
  public static void main(final String[] args) {
    TreeSet<String> set = new TreeSet<String>();
    set.add("CCCCC");
    set.add("BBBBB");
    set.add("DDDDD");
    set.add("BBBBB");
    set.add("AAAAA");

    Iterator<String> iterator = set.iterator();
    while (iterator.hasNext()) {
      System.out.print(iterator.next() + " ");
    }
  }
}
```

AAAAA BBBBB CCCCC DDDDD

Les collections de type Map : les associations de type clé/valeur

Les collections de type Map

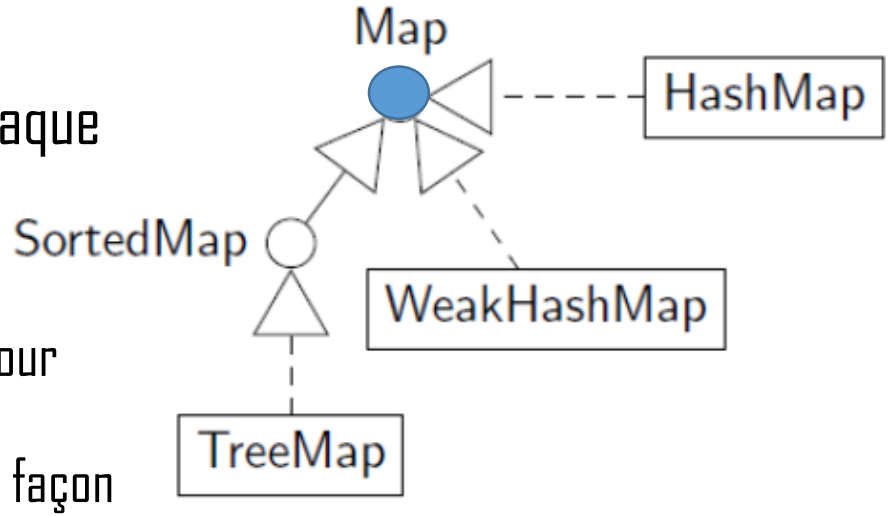
- Les collections de type **Map** sont définies et implémentées comme des dictionnaires sous la forme d'associations de paires de type **clés/valeurs**.
- La **clé** doit être **unique**. En revanche, la même valeur peut être associée à plusieurs clés différentes.
- Avant l'apparition du framework Collections, la classe dédiée à cette gestion était la classe **Hashtable**.
- Un objet de type **Map** permet de lier un objet avec **une clé qui peut être un type primitif ou un autre objet**.
- Il est ainsi possible d'obtenir un objet à partir de sa clé.

L'interface Map<K,V>

- L'interface **java.util.Map<K,V>** , ajoutée à Java 1.2, définit les fonctionnalités pour une collection qui associe des clés à des valeurs.
- Chaque clé ne peut être associée qu'à une seule valeur. Chaque **clé** d'une Map doit être **unique**.

Le langage vous propose trois implémentations de bases :

- **HashMap<K,V>** : implémentation utilisant une table de hachage pour stocker ses éléments;
- **TreeMap<K,V>** : implémentation qui stocke les éléments triés, de façon naturelle par défaut, mais utilisable avec un comparateur ;
- **LinkedHashMap<K,V>** : implémentation qui combine table de hachage et liens chaînés pour stocker ses éléments, ce qui facilite leur insertion et leur suppression.



L'interface Map<K,V> : Méthodes

Méthode	Rôle
void clear()	Supprimer tous les éléments de la collection
boolean containsKey(Object)	Indiquer si la clé est contenue dans la collection
boolean containsValue(Object)	Indiquer si la valeur est contenue dans la collection
Set entrySet()	Renvoyer un ensemble contenant les paires clé/valeur de la collection
Object get(Object)	Renvoyer la valeur associée à la clé fournie en paramètre
boolean isEmpty()	Indiquer si la collection est vide
Set keySet()	Renvoyer un ensemble contenant les clés de la collection
Object put(Object, Object)	Insèrer la clé et sa valeur associée fournies en paramètres
Object remove(Object)	Supprimer l'élément dont la clé est fournie en paramètre
int size()	Renvoyer le nombre d'éléments de la collection

Parcours d'un Map<K,V>

- Une collection de type Map ne propose pas directement d'Iterator sur ses éléments.
- la collection peut être parcourue de trois manières :
  - parcours de l'ensemble des clés : méthode **keySet()**
  - parcours des valeurs : méthode **values()**
  - parcours d'un ensemble de paires clé/valeur : interface interne **Map.Entry<K,V>**

Parcours des clés : méthode **keySet()**

- La méthode **keySet()** permet d'obtenir un ensemble contenant toutes les clés.

```
Map<Integer, String> hm = new HashMap<>();
      hm.put(10, "1");
      hm.put(20, "2");
      hm.put(30, "3");
      hm.put(40, "4");
      hm.put(50, "5");
      //Ceci va écraser la valeur 5
      hm.put(50, "6");

     Set s= hm.keySet();
     Iterator i1=s.iterator();
     while(i1.hasNext())
     System.out.println (i1.next());
```

```
50
20
40
10
30
```

Parcours des valeurs : méthode **values()**

- La méthode **values()** permet d'obtenir une **collection** contenant toutes les valeurs.
- La **valeur de retour** est une **Collection** et non un ensemble car il peut y avoir des doublons (plusieurs clés peuvent être associées à la même valeur).

```
Map<Integer, String> hm = new HashMap<>();

      hm.put(10, "1");
      hm.put(20, "2");
      hm.put(30, "3");
      hm.put(40, "4");
      hm.put(50, "5");
      //Ceci va écraser la valeur 5
      hm.put(50, "6");

      Collection c=hm.values();
      Iterator i1=c.iterator();
      while(i1.hasNext())
      System.out.print (" "+i1.next());
```

```
6 2 4 1 3
```

parcours d'un ensemble de paires clé/valeur

- La méthode **entrySEt ()** retourne une collection de type **Set** qui sera définit ainsi **Set<Entry<k,v>>.**
- Ce sera donc une collection d'objets qui contiendra tous les couples clé-valeur de notre Map.
- Cet objet est une classe interne à l'interface Map et contient quelques méthodes utiles pour récupérer les informations.

parcours d'un ensemble de paires clé/valeur
Interface *interne* **Entry<K,V>** de **Map**

- L'interface **Map<K,V>** contient l'interface interne **public Map.Entry<K,V>** qui correspond à un couple clé-valeur
- Cette interface contient 3 méthodes
  - **K getKey()**
  - **V getValue()**
  - **V setValue(V valeur)**
- La méthode **entrySet()** de **Map** renvoie un objet de type « ensemble (**Set**) de **Entry** »

Interface *interne* Entry<K,V> de Map : Exemple

```
Map<Integer, String> hm = new HashMap<>();

System.out.println("Parcours de l'objet HashMap : ");
      Set<Entry<Integer, String>> setHm = hm.entrySet();
      Iterator<Entry<Integer, String>> it = setHm.iterator();
      while(it.hasNext()){
         Entry<Integer, String> e = it.next();
         System.out.println(e.getKey() + " : " + e.getValue());
      }

      System.out.println("Valeur pour la clé 8 : " + hm.get(8));
```

```
50 : 6
20 : 2
40 : 4
10 : 1
30 : 3
Valeur pour la clé 8 : null
```

La classe HashMap & LinkedHashMap<K,V>

- ces deux objets sont deux des plus utilisés.
- Leur principale différence réside dans le fait que LinkedHashMap gère en plus des liens chaînés
- LinkedHashMap permet de parcourir les éléments de la collection **dans l'ordre d'insertion** alors que HashMap ne le permet pas

HashMap & LinkedHashMap<K,V> : Exemple

```
Map<Integer, String> lhm = new LinkedHashMap<>();
      lhm.put(10, "1");
      lhm.put(20, "2");
      lhm.put(30, "3");
      lhm.put(40, "4");
      lhm.put(50, "5");

      System.out.println("Parcours de l'objet LinkedHashMap : ");
      Set<Entry<Integer, String>> setLhm = lhm.entrySet();
      Iterator<Entry<Integer, String>> it2 = setLhm.iterator();
      while(it2.hasNext()){
         Entry<Integer, String> e = it2.next();
         System.out.println(e.getKey() + " : " + e.getValue());
```

```
Parcours de l'objet LinkedHashMap :
10 : 1
20 : 2
30 : 3
40 : 4
50 : 5
```

Trier une collection

Les classes utilitaires

- **Collections** (avec un *s* final) fournit des méthodes **static** pour, en particulier,
  - trier une collection
  - faire des recherches rapides dans une collection triée
- **Arrays** fournit des méthodes **static** pour, en particulier,
  - Trier,
  - faire des recherches rapides dans un tableau trié
  - transformer un tableau en liste

Trie d'une collection

- Afin de trier **une liste, :**

Collections.sort(l);

- Pour que La liste soit correctement triée, il faut que les éléments de la liste soient *comparables*
- Plus exactement, la méthode **sort()** ne fonctionnera que si tous les éléments de la liste sont d'une classe qui implante l'interface

java.lang.Comparable<? super E>

Interface **Comparable<T>**

- Cette interface correspond à l'implantation **d'un ordre naturel** dans les instances d'une classe
- Elle ne contient qu'une seule méthode : **int compareTo(T t)**
- Cette méthode renvoie
  - un entier positif si l'objet qui reçoit le message est plus grand que **t**
  - 0 si les 2 objets ont la même valeur
  - un entier négatif si l'objet qui reçoit le message est plus petit que **t**

Interface **Comparable<T>**

- Toutes les classes du JDK qui enveloppent les types primitifs (**Integer** par exemple) implantent l'interface **Comparable**
- Il en est de même pour les classes du JDK : **String**, **Date**, **Calendar**, **BigInteger**, **BigDecimal**, **File**, **Enum** et quelques autres
- Par exemple, **String** implémente **Comparable<String>**

Problème

- 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** ?

Solution

- on construit un objet qui sait comparer 2 éléments de la collection

  (interface **java.util.Comparator<T>**)

- On passe cet objet en paramètre à la méthode **sort**

Interface Comparator

- Elle propose une seule méthode : **int compare(Object o1, Object o2)**
- Elle doit renvoyer :
  - Un entier positif si o1 est "plus grand" que o2.
  - 0 si la valeur de o1 est identique à celle de o2.
  - Une entier négatif si o1 est "plus petit" que o2.

Interface Comparator : Exemple

```
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; }
```

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

Bibliographie

- https://www.jmdoudoux.fr/java/dej/chap-collections.htm
- https://openclassrooms.com/courses/java-et-les-collections
- Les collections en Java, L. Nerima
- Collections, Sophia Antipolis, Université de Nice
- Collections dans Java, Olivier Curé