Chapitre 7- Collection

Objectifs

- Les collections
- La classe Collections
- Les interface List et Set
- Les méthodes qui manipulent les collections
- ArrayList
- LinkedList
- HashSet

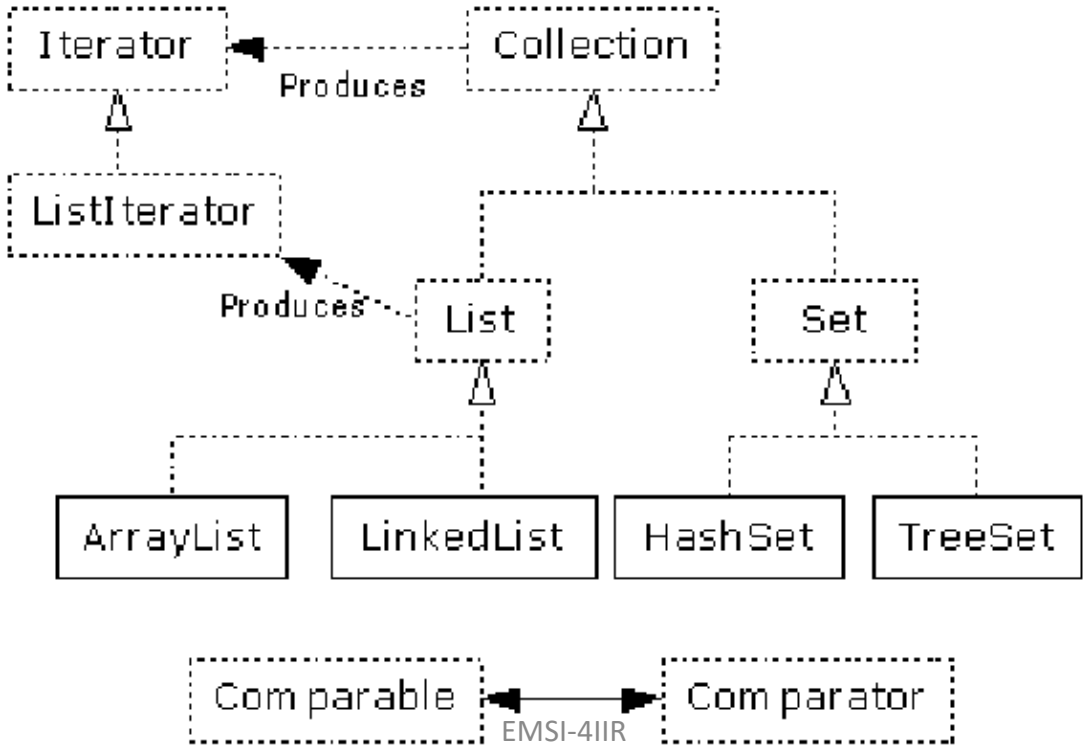
Collection de données

- une collection de données regroupe un ensemble d'objet de même type dans une instance (objet) d'une classe qui possède un protocole particulier pour l'ajout, le retrait et la recherche d'éléments
- Exemple : pile, file d'attente, séquence d'éléments.
- Les classes collection sont définies dans le package : **java.util** ;
- Le JDK définie les collections à partir de deux Interfaces Java :
  - Collection
  - Map

Collection : Interfaces

Les Collections proposent deux types d'interface:

- Interface Collection : Collection<E>
- Interface Map: Map<K,V>



Collection: méthodes communes

`boolean add(Object)` : ajouter un élément
`boolean addAll(Collection)` : ajouter plusieurs éléments
`void clear()` : tout supprimer
`boolean contains(Object)` : test d'appartenance
`boolean containsAll(Collection)` : appartenance collective
`boolean isEmpty()` : test de l'absence d'éléments
`Iterator iterator()` : pour le parcours
`boolean remove(Object)` : retrait d'un élément
`boolean removeAll(Collection)` : retrait de plusieurs éléments
`boolean retainAll(Collection)` : intersection
`int size()` : nombre d'éléments
`Object[] toArray()` : transformation en tableau
`Object[] toArray(Object[] a)` : tableau de même type que a

Collection: Caractéristiques

- Ordre sur les éléments?
  - Collection Ordonnée
  - Collection non Ordonnée
- Doublons autorisés ou non
  - ***liste*** (`List`) : avec doubles
  - ***ensemble*** (`Set`) : sans doubles
- L'accès
  - Par indexe
  - D'une manière séquentielle

Collection : Collection<E>

L'interface Collection correspond à un objet qui contient un groupe d'objets de type E;

L'interface Collection<E> est l'interface mère d'un ensemble de classes abstraites et de classe ordinaire : AbstractCollection, AbstractList, AbstractQueue, AbstractSequentialList, AbstractSet, ArrayDeque, ArrayList, AttributeList, CopyOnWriteArrayList, CopyOnWriteArraySet, DelayQueue, EnumSet, HashSet, LinkedBlockingDeque, LinkedHashSet, LinkedList, PriorityQueue, RoleList, Stack, SynchronousQueue, TreeSet, Vector…

Nous verrons les classes collection suivantes:

- ArrayList;
- LinkedList ;
- HashSet.

La classe Collections:

- Cette classe compte une cinquantaine de méthodes très utiles pour la manipulations des collections.

- Certaines méthodes permettent de construire des listes ou des tables particulières (vides, ne comportant qu'un unique élément), synchronisés ou immutables .

- Certaines méthodes permettent de mélanger les éléments d'une collection, soit de façon prévisible (rotate), soit de façon aléatoire (shuffle).

- On trouve encore des méthodes permettant de faire du tri, de rechercher le nombre de fois qu'un élément particulier apparaît dans une collection, ou encore de rechercher le plus petit ou le plus grand élément d'une collection.

Collections: Classe

binarySearch : Recherche un élément dans une liste;

checkedList: Retourne une liste typée dynamiquement avec le type spécifié en deuxième paramètre.

copy: Copie la liste source dans la liste de destination.

Fill: Remplace tous les éléments de la liste par l'élément passé en deuxième paramètre.

indexOfSubList: Retourne la position de départ (première occurence) de la première occurence de la sous-liste passée en paramètre. Cette méthode renverra -1 si aucun élément n'a été trouvé.

lastIndexOfSubList: Identique à la méthode ci-dessus mais retourne l'index de la dernière occurence. Renvoie -1 si aucun élément n'a été trouvé.

replaceAll: Remplace toutes les occurrences d'une certaine valeur par une autre.

reverse: Renverse l'ordre des éléments de la liste.

rotate: Opère une rotation entre les éléments d'une liste à une certaine distance.

shuffle: Permute de façon aléatoire les éléments de la liste.

swap: Échange les éléments de la liste aux positions spécifiées.

Collections

```
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);

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

ArrayList

Définition

ArrayList est Un tableau dynamique : un tableau dont la taille (nombres d'éléments) est dynamique (qui varie en fonction des objets ajoutés/supprimés);

Caractéristiques :

- Accéder à un élément d'indice i est instantané (directe: ième cellule du tableau)
- Ajouter/supprimer un élément nécessite le décalage des éléments suivants;

Méthode ArrayList.add

```
public static void main(String args[]) {
        ArrayList<String> liste = new ArrayList<>();
        liste.add("A"); liste.add("B");
        liste.add("C"); liste.add(0, "D");
        System.out.println(liste);
        // [D, A, B, C]
    }
```

Méthode ArrayList.remove

- La méthode public void remove(int) supprime un élément d'une ArrayList à un emplacement spécifié;

```
public static void main(String args[]) {
        ArrayList< Integer> liste = new ArrayList<
Integer>();
        liste.add(4);liste.add(5);
        liste.add(2);
        System.out.println(liste); // [4, 5, 2]
        liste.remove(1);
        System.out.println(liste); //[4, 2]
    }
```

Méthode ArrayList.set

- La méthode public **public void set(int, Object)** permet de modifier un élément à un emplacement spécifié dans une ArrayList;

```
public static void main(String args[]) {
        ArrayList< Integer> liste = new ArrayList< Integer>();
        liste.add(4); liste.add(5);
        liste.add(2);
        System.out.println(liste); //[4, 5, 2]
        liste.set(1, 15);
        System.out.println(liste); //[4, 15, 2]
    }
```

Méthode ArrayList. get

- La méthode public **Object get(int)** permet de récupérer un élément d'un emplacement spécifié dans une ArrayList;

```
public static void main(String args[]) {
        ArrayList<Integer> liste = new ArrayList<
Integer>();
        liste.add(4); liste.add(5);
        liste.add(2);
        System.out.println(liste); //[4, 5, 2]
        System.out.println(liste.get(1)); //4
    }
```

Méthode ArrayList.size

- La méthode public **int size()** permet de renvoyer la taille actuelle de ArrayList;

```
public static void main(String args[]) {
        ArrayList< Integer> liste = new
ArrayList< Integer>();
        liste.add(4); liste.add(5);
        liste.add(2);
System.out.println("la taille est :" +
liste.size());
        // la taille est : 3
    }
```

Méthode Collections.sort

- La méthode public **int size()** permet de renvoyer la taille actuelle de ArrayList;

```
public static void main(String args[]) {
        ArrayList< Integer> liste = new ArrayList<
Integer>();
        liste.add(4);liste.add(5);
        liste.add(2);
        System.out.println("Liste non triée : " + liste);
        //Liste non triée : [4, 5, 2]
        Collections.sort(liste);
        System.out.println("liste triée : " + liste);
        //liste triée : [2, 4, 5]
    }
```

ArrayList : Exercice

- **Exercice:**
  1. Créer une collection (ArrayList) de noms de pays
  2. alimenter cette collection avec quelques valeurs
  3. afficher la taille de la collection
  4. Créer une méthode qui affiche le contenu de la collection. Vérifier la taille de la collection, si elle est vide afficher un message d'erreur.
  5. vider la collection
  6. alimenter de nouveau la liste de pays;
  7. Rechercher des éléments dans la liste ;
  8. Supprimer un élément ;
  9. Trier la collection et réafficher la liste des pays.

LinkedList

Définition

Une liste chainée est une liste, dont les éléments sont organisés de manière similaire a une chaîne. Chaque élément de la liste (i.e. un maillon) est relié a un autre, de sorte que la liste forme une chaine d´éléments.

Iterator : permet d'obtenir un itérateur sur la collection (pour récupérer ses éléments un par un)

LinkedList

Caractéristiques

- Liste chaînée avec un double maillage (précédent, suivant)
- accès aux éléments lourd
- permettent d'implanter les structures FIFO (file) et LIFO (pile)
- méthodes supplémentaires : addFirst(), addLast(), getFirst(), getLast(), removeFisrt(), removeLast();
- Maintien d'une référence vers le premier et dernier maillon;
- Ajouter ou supprimer un élément en tête ou fin de liste est en temps constant;
- Accéder à un élément d'indice i nécessite de parcourir depuis le début ou la fin les maillons jusqu'à i;
- Gourmande en mémoire (une référence précédent, une référence suivant et une référence maillon pour chaque valeur);
- Dispersion en mémoire.

LinkedList

```
// créer une liste chaînée (LinkedList)
        LinkedList< String> link = new LinkedList< String>();
         // ajouter des éléments à LinkedList
        link.add("Mostafa");        link.add("Ismail");
        link.add("Dounia");         link.add("Badr");
        link.add("Omar");         link.addLast("Mohamed");
        link.addFirst("Fatima");    link.add(1, "Haitam");
// supprimer des éléments de la liste
        link.remove("Badr");       link.remove(2);
// supprimer le premier et le dernier élément
        link.removeFirst();       link.removeLast();
// obtenir et définir une valeur
        Object val = link.get(2);
        link.set(2, (String) val + "est modifié");
```

LinkedList: Exercices

Exercice-1:

1. Créer une collection **ListeDesNoms** de type **LinkedList** ;
2. alimenter cette collection avec les valeurs suivante : Ayoub, Sanae, Ali, Asmaa et Noura ;
3. afficher la collection ;
4. supprimer le dernier élément de la liste
5. Ajouter un élément en tête de la liste
6. Renvoyer le premier élément de la liste

LinkedList: Exercices

Exercice-2:

1. Créer une collection **weekDaysList** de type **LinkedList** ;
2. Créer un Tableau **weekDays** contenant les jours de la semaine ;
3. Insérer tous les éléments du tableau **weekDays** dans **weekDaysList ;**
4. Parcourir la liste **weekDaysList** et afficher tous ces éléments ;

Set

- Set est une interface en java
- Set est une collection non ordonnée d’éléments de type E.
- Set est une collection qui ne contient pas d'éléments dupliqués. Autrement dit : un Set ne contient pas un pair d'objets e1 et e2 tel que e1.equals(e2) renvoie vrai;
- Set peut contenir au plus un élément null;
- La classe HashSet implémente l'interface Set

Set

Problème: comme deux objets distincts ont des références différentes, on ne pourra jamais avoir deux objets égaux même si toutes leurs valeurs sont identiques

Solution: Il faudra définir un comparateur qui sera capable de tester l'égalité de deux objets (equals et compareTo)

Remarque : Même s'il n'existe pas d'ordre dans une collection, l'implémentation s'appuie sur une organisation des éléments afin de garantir un accès efficace.

HashSet

- Un type prédéfinit qui implémente le Set avec une table hachage;
- Afin de bien utiliser le HashSet, L'utilisateur devra définir les méthodes **hashCode** et **equals** dans la classe des éléments **E**;

Exemple :

```
// ensemble vide
HashSet<E> e1 = new HashSet<E> ();

/* ensemble contenant tous les éléments de la collec-
tion c */
  HashSet<E> e2 = new HashSet<E> (c);
```

HashSet : Méthodes usuelles

add : ajout d'un élément s'il n'appartient pas encore à l'ensemble

```
HashSet<E> e = new HashSet<E> ();
E elem = new E();
boolean nouveau = e.add(elem);
// true si elem a été ajouté
if (nouveau) System.out.println(elem + "ajouté à l'ensemble");
else System.out.println(elem + "existait déjà dans
l'ensemble");
```

contains: test d'appartenance

remove: suppression d'un élément

```
boolean appartient = e.contains(elem);
//  elem appartient-il à e ?
boolean trouve = e.remove(elem);
//false si elem n'appartient pas à e
```

HashSet - parcours

Parcours à l'aide d'un **itérateur**

```
HashSet<E> e = new HashSet<E> ();
... // code d'ajouts d'éléments à l'ensemble
Iterator <E> iter = e.iterator();
while (iter.hasNext()) {
     E elem = iter.next();
     System.out.println(elem);
```

- Remarques :
  - Les éléments d'un ensemble n'étant pas ordonnées, aucun ordre d'itération n'est assuré
  - L'ordre d'itération peut varier dans le temps

HashSet

```
HashSet< String> pays = new HashSet< String>();

pays.add("Maroc");pays.add("Tunisie"); pays.add("Algérie");
pays.add("Espagne");pays.add("Malaisie");

pays.add("Espagne"); // ajouter un élément en double

System.out.println("Est ce que la tunisie appartient aux pays :"
+ pays.contains("Tunisie"));


pays.remove("Espagne");

// Suppression d'éléments de HashSet à l'aide de remove ()

System.out.println("Liste après avoir retiré l'espagne:" + pays);

// Itérer sur HashSet

      Iterator< String> it = pays.iterator();

            while (it.hasNext())

           System.out.println(it.next());

}
```

Recherche d'un élément

- Méthode
  - `public boolean contains(Object o)`
- Utilise l'égalité entre objets
  - égalité définie par `boolean equals(Object o)`
  - par défaut (classe `Object`) : égalité de références
  - à redéfinir dans chaque classe d'éléments

Tri d'une structure collective

- Algorithmes génériques
  - Collections.sort(List l)
  - Arrays.sort(Object[] a,…)
- Condition : collection d'éléments dont la classe définit des règles de comparaison
  - en implémentant l'interface **`java.lang.Comparable`**
    - implements Comparable
  - en définissant la méthode de comparaison
    - public int compareTo(Object o)
      - **`a.compareTo(b) == 0`** si **`a.equals(b)`**
      - **`a.compareTo(b) < 0`** pour **`a`** strictement inférieur à **`b`**
      - **`a.compareTo(b) > 0`** pour **`a`** strictement supérieur à **`b`**