Les collections

AMMOR Fatimazahra




Les collections

- Les tableaux ne peuvent pas répondre à tous les besoins de stockage et surtout ils manquent de fonctionnalités. La large diversité d'implémentations proposées par l'API Collections de Java permet de répondre à la plupart des besoins.
  - Avantages: (1) Un tableau connait bien le type des données manipulées, (2) une vérification du type au moment de la compilation, (3) Un tableau connait bien sa taille, (4) Un tableau peut manipuler les primitives directement.
  - Inconvénients: (1) Un tableau peut manipuler un seul type d'objets, (2) Un tableau dispose d'une taille fixe.

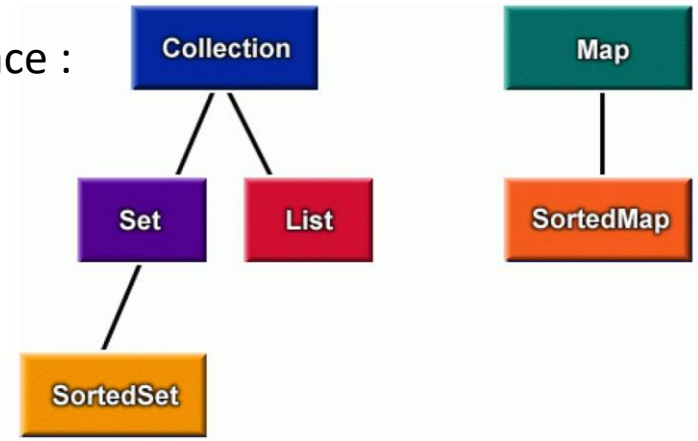
```
// Declare a table of integers with 5 elements
int[] myArray = new int[5];
// Initialize the table with values myArray[0] = 10; myArray[1] = 20; myArray[2] = 30; myArray[3] = 40; myArray[4] = 50;
// Alternatively, initialize the table with a list of values int[] myArray2 = {10, 20, 30, 40, 50};
// Declare a new table with the same size as the original
int[] newArray = new int[myArray.length];
// Copy the elements from the original table to the new table
System.arraycopy(myArray, 0, newArray, 0, myArray.length);
```

```
// Search for a specific element in the tableau
int targetElement = 30;
for (int i = 0; i < myArray.length; i++) {
   if (myArray[i] == targetElement) {
      System.out.println("Found element " +
                  targetElement + " at index " + i);
      break;
   }
}
```

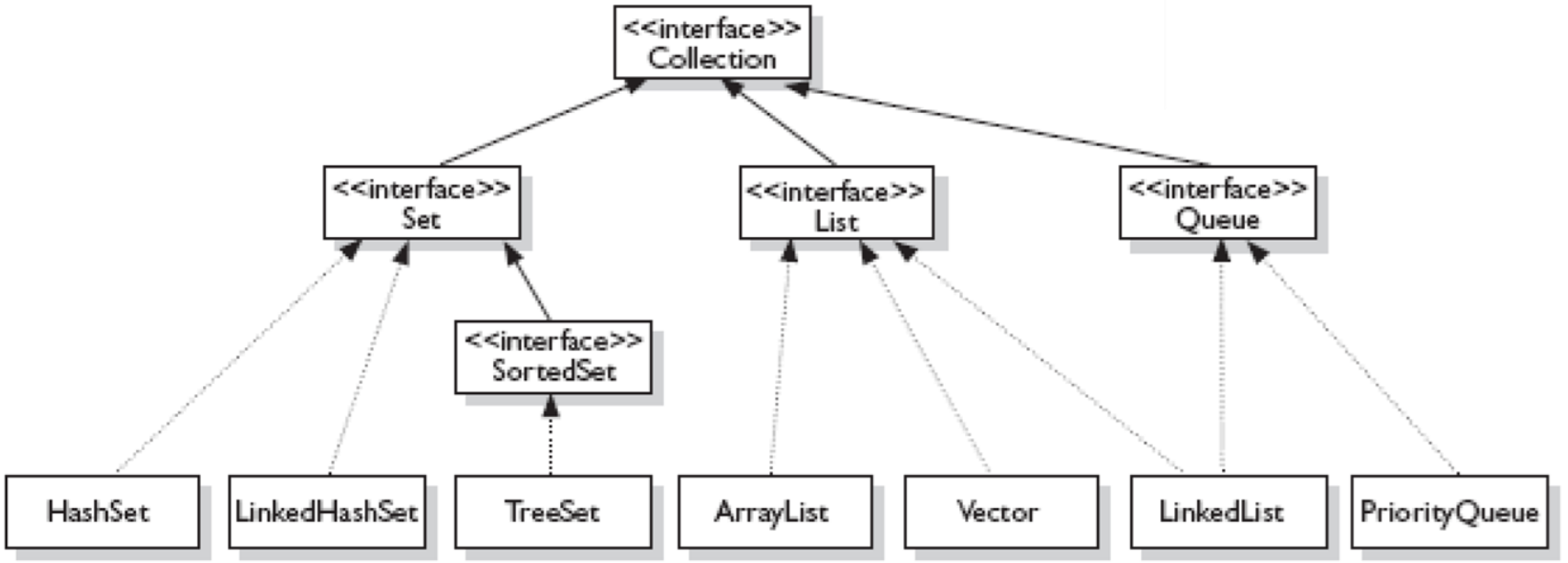
```
public class TestArgsMain {
public static void main( String [] args ) {
      int accumulator = 0;
      for( String param : args ) {
            accumulator += Integer.parseInt( param );
      }
      System.out.println( accumulator );
  }
}
```

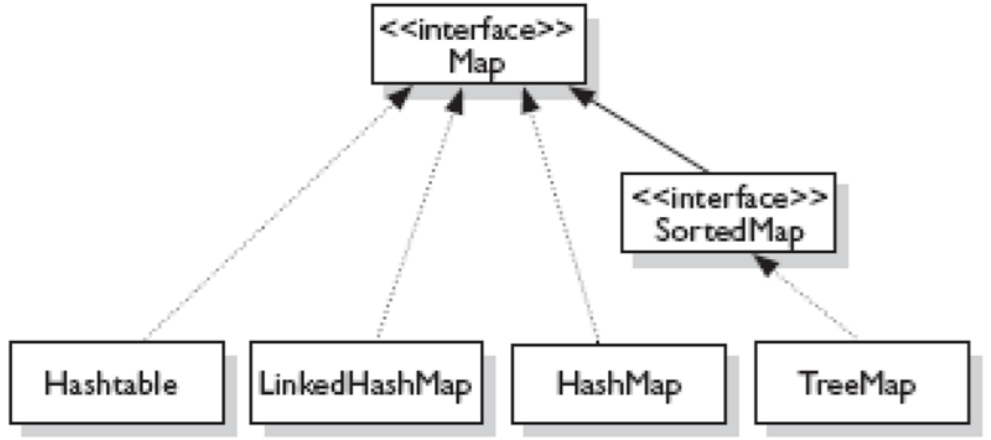
- Les **collections** sont des objets qui permettent de gérer des ensembles d'objets.
- Lors de l’utilisation d’une collection, il est possible pour nous d'ajouter, supprimer, obtenir et parcourir les éléments.
- Une collection est un groupe d’objets manipulés comme un seul objet (comme un sac).
- Une collection n’a pas de taille fixe, elle peut s’agrandir si nécessaire.
- Une collection ne manipule que des Object, et ne manipule pas de primitives.
- Une collection est hétérogène.

- L'API Collections possède deux grandes familles chacune définies par une interface :
  - java.util.Collection : pour gérer un groupe d'objets
  - java.util.Map : pour gérer des éléments de type paires de clé/valeur

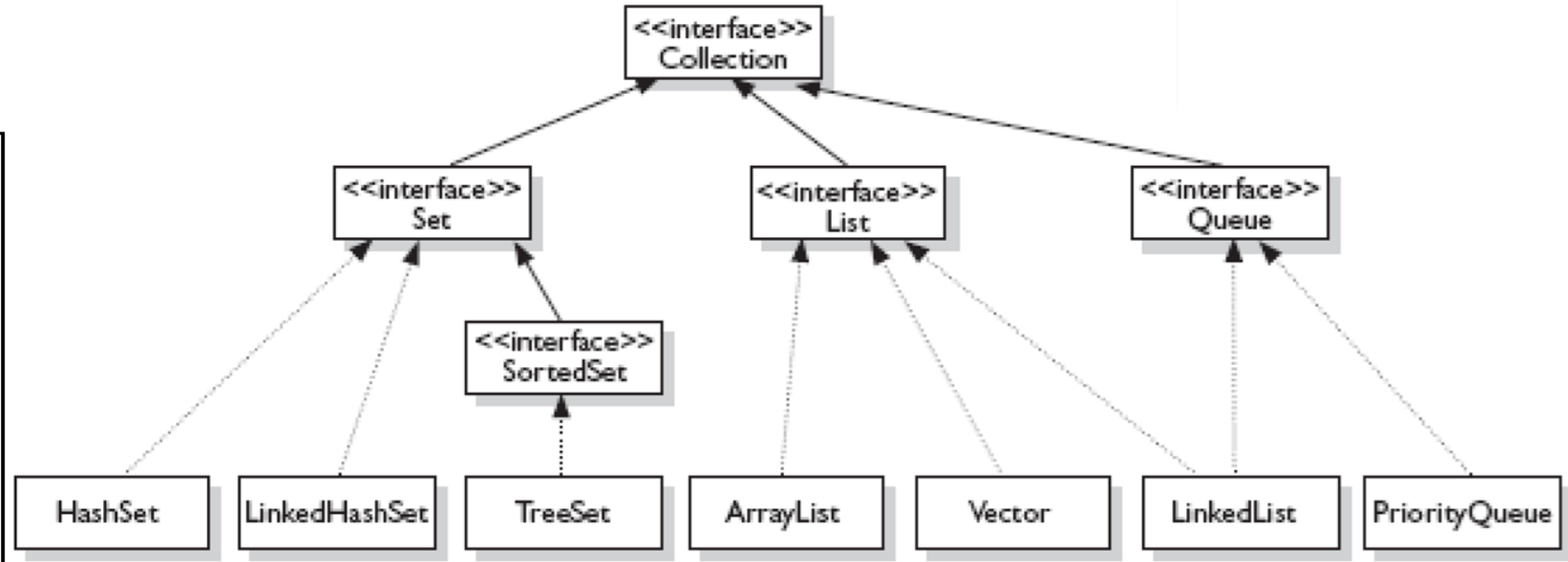


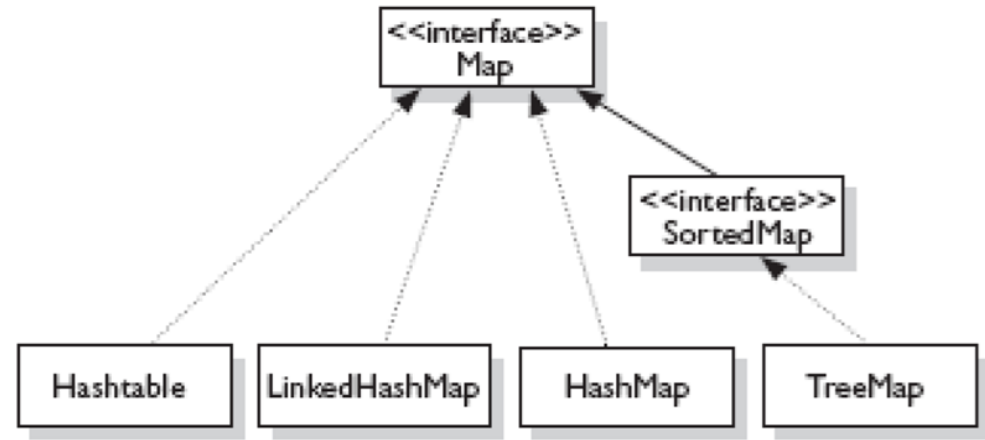
- L'API Collections propose un ensemble d'interfaces et de classes dont le but est de stocker de multiples objets.
- Elle propose quatre grandes familles de collections, chacune définie par une interface de base :
  1. List : collection d'éléments ordonnés qui accepte les doublons
  2. Set : collection d'éléments non ordonnés par défaut qui n'accepte pas les doublons
  3. Map : collection sous la forme d'une association de paires clé/valeur
  4. Queue et Deque : collections qui stockent des éléments dans un certain ordre avant qu'ils ne soient extraits pour traitement





```java
public interface Collection {

    // Basic Operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(Object element);
    boolean remove(Object element);

    Iterator iterator();

    // Bulk Operations
    boolean containsAll(Collection c);
    boolean addAll(Collection c);
    boolean removeAll(Collection c);
    boolean retainAll(Collection c);
    void clear();

    // Array Operations
    Object[] toArray();
    Object[] toArray(Object a[]);

}
```

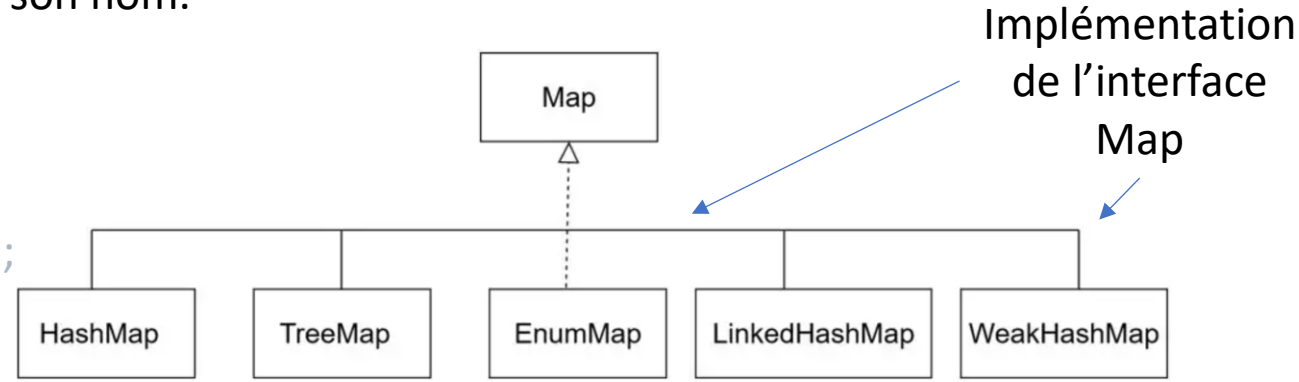




HashMap

HashMap = Clé unique.
Créer un carnet d'adresses simple où chaque personne ajoutée sera identifiée par son nom (la clé) et son numéro de téléphone (la valeur). Le HashMap est idéal pour cette tâche car il nous permet d'accéder rapidement au numéro de téléphone d'une personne en connaissant son nom.

```
import java.util.Map
// implementation de Map avec HashMap
Map<String, String> carnetAdresse = new HashMap<>();
```



- Enregistrer vos contacts : les clés sont des String (les noms) et les valeurs sont également des String (les numéros de téléphone).
- Ajouter des contacts à votre carnet. On utilise pour cela la méthode put().
- Récupérer le numéro de téléphone d'une personne. On utilise la méthode get().
- Vérifier si un contact existe dans le carnet. La méthode containsKey().
- Supprimer un contact du carnet.

```
for (Map.Entry e : carnet.entrySet()) {
System.out.println("clé: "+e.getKey() + ", valeur: " + e.getValue());
}
```

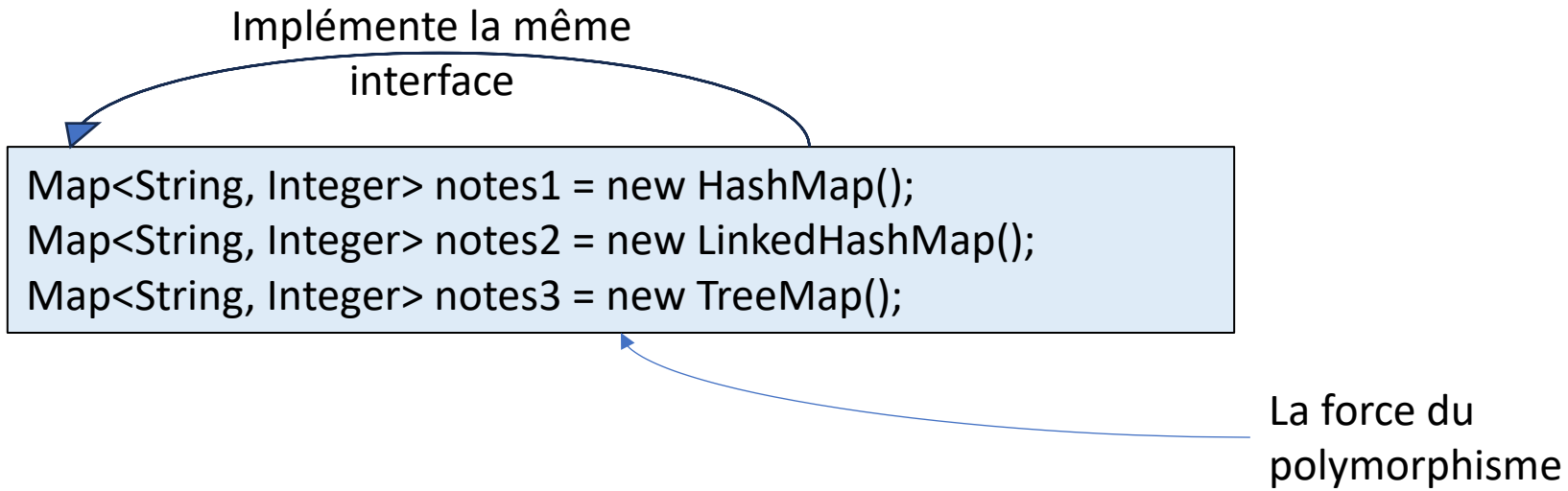
- **put(K, V) :** Insère l'association d'une clé K et d'une valeur V dans le Map. Si la clé est déjà présente, la nouvelle valeur remplace l'ancienne.
- **putAll(Map)** : Insère toutes les entrées (clés/valeurs) du Map fourni en paramètre.
- **putIfAbsent(K, V)** : Insère l'entrée (K, V) si la clé K n'est pas déjà associée à la valeur V.
- **get(K)**: Retourne la valeur associée à la clé K spécifiée. Si la clé n'est pas trouvée, elle retourne null.
- **getOrDefault(K, defaultValue)** : Retourne la valeur associée à la clé K spécifiée. Si la clé n'est pas trouvée, elle retourne la valeur par défaut.
- **containsKey(K)** : Vérifie si la clé spécifiée K est présente dans le Map ou non.
- **containsValue(V)** : Vérifie si la valeur spécifiée V est présente dans le Map ou non.
- **replace(K, V)** : Remplace la valeur de la clé K par la nouvelle valeur spécifiée V.
- **replace(K, oldValue, newValue)** : Remplace la valeur de la clé K par la nouvelle valeur *newValue* seulement si la clé K est associée à la valeur *oldValue*.
- **remove(K)** : Supprime l'entrée du Map représentée par la clé K.
- **remove(K, V)** : Supprime l'entrée du Map représentée par la clé K associée à la valeur V.
- **keySet()** : Retourne l'ensemble de toutes les clés présentes dans une map.
- **values()** : Retourne un ensemble de toutes les valeurs présentes dans une map.
- **entrySet()** : Retourne un ensemble de toutes les correspondances clé/valeur présentes dans une map.

```
import java.util.HashMap;
import java.util.Map;

public class CarnetAdresses {
    public static void main(String[] args) {
        // Création d'un HashMap pour stocker les contacts
        Map<String, String> carnet = new HashMap<>();

        // Ajout de contacts
        carnet.put(« Amine", "0123456789");
        carnet.put(« Ali", "9876543210");
        carnet.put(« Anass", "5551212");
        carnet.put(« Khadija", "5551212");
        carnet.put(« Aicha", "5551212");
```

```
        // Récupération du numéro de téléphone de Anass
        String numeroAnass = carnet.get(« Anass");
        System.out.println("Le numéro de Anass est : " +
numeroAnass);

        // Vérification si un contact existe
        if (carnet.containsKey(«Khadija")) {
            System.out.println(" Khadija est dans le carnet");
        } else {
            System.out.println(" Khadija n'est pas dans le carnet");
        }

        // Supprimer un contact
        carnet.remove(«Aicha»);
    }
}
```

LinkedHashMap/TreeMap

- HashMap = ordre aléatoire
- LinkedHashMap = HashMap + garde le même ordre d'insertion
- TreeMap = HashMap + ordre naturel (alphabétique, numérique, etc.)



TreeMap

firstKey()
Renvoie la première clé (la plus petite dans l'ordre).

lastKey()
Renvoie la dernière clé (la plus grande dans l'ordre).

headMap(K toKey) et headMap(K toKey, boolean inclusive)
Renvoie une sous-map contenant toutes les entrées dont les clés sont **strictement inférieures** ou (optionnellement) inférieures ou égales à toKey.

tailMap(K fromKey) et tailMap(K fromKey, boolean inclusive)
Renvoie une sous-map contenant toutes les entrées dont les clés sont **supérieures ou égales** ou (optionnellement) strictement supérieures à fromKey.

subMap(K fromKey, K toKey) et subMap(K fromKey, boolean fromInclusive, K toKey, boolean toInclusive)
Renvoie une sous-map avec les entrées dont les clés sont comprises dans un intervalle donné.

Comme TreeMap implémente NavigableMap, il dispose de méthodes supplémentaires pour la navigation.

```
System.out.println(map.headMap(3)); // {1=A, 2=B}
System.out.println(map.headMap(3, true)); // {1=A, 2=B, 3=C}
```

```
System.out.println(map.tailMap(2)); // {2=B, 3=C}
System.out.println(map.tailMap(2, false)); // {3=C}
```

```
System.out.println(map.subMap(1, 3)); // {1=A, 2=B}
System.out.println(map.subMap(1, true, 3, true)); // {1=A, 2=B, 3=C}
```

`lowerKey(K key)`	Renvoie la clé immédiatement inférieure à `key`.
`floorKey(K key)`	Renvoie la plus grande clé inférieure ou égale à `key`.
`ceilingKey(K key)`	Renvoie la plus petite clé supérieure ou égale à `key`.
`higherKey(K key)`	Renvoie la clé immédiatement supérieure à `key`.
`descendingMap()`	Renvoie une vue inversée du `TreeMap`.

ArrayList

- **ArrayList** = tableau dynamique.
- Ne tiens pas en compte l'ordre d'ajout
- La capacité initial d'un arrayList est 10
- Créer un arrayList pour enregistrer les produits d'un magasin en ligne

```
ArrayList<String> listProduit = new ArrayList<>(List.of("Pates", "Lentilles", "Pain", "Eau", "Riz", "Chocolat"));
```

  - Ajouter des produits, supprimer des produits, vérifier la présence d'un produit, retourner le nombre de produits dans la liste, afficher le produit numéro 5 dans la liste, remplacer le produit 4 par « Lait ».
- Créer un linkedList pour enregistrer les produits d'une commande d'un client
  - Ajouter les produits existants dans votre linkedList, vérifier la présence d'un produit, insérer un produit dans le début de la liste (**addFirst**), supprimer le produit (**removeFirst**), afficher le dernier produit
  - Tenter de créer des doublons, et supprimer les doublons en transformant votre Linkedlist en un HashSet.

```
List<Integer> nombres = Arrays.asList(1, 2, 3, 2, 1, 4);
Set<Integer> nombresSansDoublons = new HashSet<>(nombres);
System.out.println(nombresSansDoublons);
```

ArrayList

- **ArrayList** = tableau dynamique.
- Ne tiens pas en compte l'ordre d'ajout
- La capacité initial d'un arrayList est 10

```
listColl.addAll(3, l1);   // Ajoute une collection au milieu
listColl.clear();    //Supprimer tous les éléments
boolean b = listColl.contains(o);     // La valeur de b vaut true si l'objet o
                                          appartient à la liste.
system.out.println(listColl.get(2));     // retourne l'objet à la position 2
boolean b = listColl.isEmpty();    // Retourne true si la liste est vide.
boolean b = listColl.remove("o4");   // Retourne true si l'objet existe et a été
                                          supprimé avec succès.
list.removeAll(listColl);     // Elle cherche les éléments et les supprime
list.set(3, "o5");      // L'objet dans la position 3 a été remplacé par o5.
```

```
ArrayList l1 = new ArrayList();
ArrayList list = new ArrayList();
l1.add("mot");    l1.add(12);     l1.add(10.4f);
list.addAll(l1);
list.size() // retourne le nombre d'éléments.
list.subList(int debut, int fin) // Retourne le fragment situé entre le début et la fin.
String[] t = list.toArray(); // Retourne un tableau d'une dimension.
```

```
        for(int i = 0; i < list.size(); i++)
                system.out.println(list.get(i));
Ou par exemple si on connait le type:
        for(Integer nombre : list)
                system.out.println(nombre);
Ou en utilisant iterator:
        Iterator itr = list.iterator();
        while(itr.hasNext())
                system.out.println(itr.next());
```

```
interface Iterator {
boolean hasNext();
Object next;
void remove();
}
```

LinkedList

```
LinkedList linkedlist = new LinkedList();
ArrayList arraylist= new ArrayList();
arraylist.add("123");
arraylist.add("456");
linkedlist.addAll(arraylist);
linkedlist.add(3, "position3");
linkedlist.addAll(3, arraylist);
linkedlist.clear();
System.out.println("linkedList: "+ linkedlist);
Object str= linkedlist.clone();   // retourne une copie de la liste
System.out.println("str: "+str);
```

```
Boolean var = linkedlist.contains("String");    //vérifie si l'objet est présent dans la liste. Si l'élément existe, elle retourne true sinon false
linkedlist.indexOf(Object o)   // retourne l'indice d'un objet donné.
 linkedlist.lastIndexOf(Object o)  //retourne l'indice de la dernière occurrence d'un objet donné.
```

```
linkedlist.addFirst("string");  // insère un élément dans la première position.
Object e = linkedlist .getLast();  // retourne l'élément à la dernière position.
linkedlist.removeFirst(); // supprime l'élément de la première position.
```

Set = Interface

- HashSet = Pas de doublons : Ne tiens pas en compte l'ordre (comme le HashMap)
- LinkedHashSet = Tiens en compte l'ordre d'ajout
- TreeSet = Tiens en compte l'ordre naturel

Gestion d'une bibliothèque :

1. Créer un set qui contient les livres d'une bibliothèque.

- Afficher les livres disponibles. Chaque livre contient un titre, un auteur et une date d'edition.
- Créer une méthode qui permet d'ajouter des livres dans la bibliothèque.
- Créer une méthode qui permet de supprimer des livres de la bibliothèque.

2. Créer un set qui permet de récupérer tous les livres de la bibliothèque afficher par ordre alphabétique. Afficher les livres de la nouvelle bibliothèque.

- Créer une méthode qui permet d'afficher le premier livre dans la bibliothèque
- Créer une méthode qui permet d'afficher le dernier livre dans la bibliothèque

3. Créer une exception personnalisée vérifiée qui permet de lancer des exceptions si on veut ajouter dans la bibliothèque un livre qui existe déjà.

```
import java.util.TreeSet;
class Livre implements Comparable<Livre> {
   // ... (implémentation de la classe Livre)

   @Override
   public int compareTo(Livre autreLivre) {
            return this.titre.compareTo(autreLivre.titre); // comparaison par titre
  }
}
public class Main {
   public static void main(String[] args) {
      TreeSet<Livre> maBibliotheque = new TreeSet<>();
      // ... (ajout de livres à la bibliothèque)

      // Affichage des livres
      for (Livre livre : maBibliotheque) { System.out.println(livre); }
    }
}
```

- **Valeur négative:** Si l'objet courant est "inférieur" à l'objet passé en paramètre.
- **Valeur nulle:** Si les deux objets sont égaux.
- **Valeur positive:** Si l'objet courant est "supérieur" à l'objet passé en paramètre.

- Le TreeSet utilise les résultats de ces comparaisons pour organiser les éléments en un arbre binaire de recherche. Cet arbre garantit que les éléments sont toujours triés selon l'ordre défini par compareTo.

```
import java.util.TreeSet;
class Livre implements Comparable<Livre> {
   // ... (implémentation de la classe Livre)

   @Override
   public int compareTo(Livre autreLivre) {
            return this.titre.compareTo(autreLivre.titre); // comparaison par titre
  }
}
public class Main {
   public static void main(String[] args) {
      TreeSet<Livre> maBibliotheque = new TreeSet<>();
      // ... (ajout de livres à la bibliothèque)

      // Affichage des livres
      for (Livre livre : maBibliotheque) { System.out.println(livre); }
    }
}
```

- Si le titre du livre courant est avant le titre de l'autre livre dans l'ordre alphabétique, compareTo retourne une valeur négative.
- Si les titres sont identiques, compareTo retourne 0.
- Si le titre du livre courant est après le titre de l'autre livre, compareTo retourne une valeur positive.

- compareTo définit l'ordre naturel des éléments dans un TreeSet.
- Le TreeSet utilise cet ordre pour organiser les éléments de manière efficace.
- En parcourant le TreeSet, les éléments sont renvoyés dans l'ordre défini par compareTo.

Queue

// First In First Out (Principe FIFO) // on suit l'ordre d'ajout

Queue<Integer> q1 = new LinkedList();

Queue<Integer> q2 = new ArrayBlockingQueue(3); *// Capacité 3 à respecter sinon exception queue remplie*

ArrayList a = new ArrayList();

Ajout éléments:

a.add(1);
q1.add(2);
q2.offer(3) *// true ou false si on peut ou non ajouter dans ma queue (ajoute si possible)*

Affichage premier élément:

q1.element() *// m'affiche le premier element*

q2.peek(); *// me donner mon premier element ou null si la queue est vide (contrairement à element())*

Suppression éléments:

a.remove(indice); *// je dois spécifier l'element à supprimer*
q1.remove(); *// je supprime le premier selon le principe FIFO,*
q2.poll(); *// supprime le premier element, pas d'indice pas de bug (pas d'exception) si la queue vide contrairement à remove*

Ajout éléments:

queue.offer(3) *// true ou false si on peut ou non ajouter dans ma queue (ajoute si possible)*

Affichage premier élément:

queue.peek(); *// me donner mon premier element ou null si la queue est vide (contrairement à element())*

Suppression éléments:

queue.poll(); *// supprime le premier element, pas d'indice pas de bug (pas d'exception) si la queue vide contrairement à remove*

```
import java.util.PriorityQueue;

public class PriorityQueueExample {
    public static void main(String[] args) {
        // Création d'une PriorityQueue pour les chaînes
        PriorityQueue<String> queue = new PriorityQueue<>();

        // Ajout de chaînes à la queue (en utilisant offer)
        queue.offer("banana");
        queue.offer("apple");
        queue.offer("orange");

        // Affichage du premier élément sans le retirer (peek)
        System.out.println("Premier élément : " + queue.peek());

        // Retrait et affichage des éléments jusqu'à ce que la
                queue soit vide (poll)
        while (!queue.isEmpty()) {
            System.out.println(queue.poll());
        }
    }
}
```

Système de gestion de tâches avec différentes priorités:

- Les tâches ont une priorité (1 = élevée, 2 = moyenne, 3 = basse).
- Les tâches avec une priorité plus élevée sont exécutées en premier (nécessité d'implémenter la méthode compareTo de l'interface Comparator).
- Une **PriorityQueue** est utilisée pour stocker et traiter les tâches en fonction de leur priorité.

```
public class Tache implements Comparable<Tache> {
   private int priority; // 1 = haute, 2 = moyenne, 3 = basse
   private String description;

   …
   @Override
   public int compareTo(Tache autre) {
            return Integer.compare(this.priority, autre priority);  // ordre croissant
    }
}
```

Ordre croissant par défaut:

- Un nombre négatif si l'objet courant est inférieur à l'objet passé en paramètre.
- Zéro si les deux objets sont égaux.
- Un nombre positif si l'objet courant est supérieur à l'objet passé en paramètre.

Correction !

```
public class Tache implements Comparable<Tache> {
   private int priority; // 1 = haute, 2 = moyenne, 3 = basse
   private String description;

   public Tache(int priority, String description) {
            this.priority = priority;
            this.description = description;
   }
   public String getDescription() {
      return description;
   }
   public int getPriority() {
      return priority;
   }
```

```
      @Override
   public String toString() {
      return " Tache{" +
            "description='" + description + '\'' +
            ", priority=" + priority +
            '}';
   }
@Override
   public int compareTo(Tache autre) {
            return Integer.compare(this.priority, autre priority);
// ordre croissant
    }

}
```

```
import java.util.PriorityQueue;
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        PriorityQueue<Tache> taskQueue = new
PriorityQueue<>();
        Scanner scanner = new Scanner(System.in);

        System.out.println("Bienvenue dans le gestionnaire de
tâches !");
        while (true) {
            System.out.println("\nMenu :");
            System.out.println("1. Ajouter une tâche");
            System.out.println("2. Afficher la prochaine tâche");
            System.out.println("3. Exécuter la prochaine tâche");
            System.out.println("4. Afficher toutes les tâches");
            System.out.println("5. Quitter");

            int choice = scanner.nextInt();
            scanner.nextLine(); // Consomme le saut de ligne
```

```
            switch (choice) {
                case 1:
                    System.out.print("Entrez la description de la tâche :");
                    String description = scanner.nextLine();
                    System.out.print("Entrez la priorité de la tâche (1 =
                            haute, 2 = moyenne, 3 = basse) : ");
                    int priority = scanner.nextInt();
                    scanner.nextLine(); // Consomme le saut de ligne
                    Tache task = new Tache(priority, description);
                    taskQueue.add(task); //La tâche ajoutée et triée
                                        automatiquement par priorité.
                    System.out.println("Tâche ajoutée : " + task);
                    break;
                case 2:
                    if (!taskQueue.isEmpty()) {
                        System.out.println("Prochaine tâche : " +
                                taskQueue.peek()); // me donner mon
                                        premier element ou null si la queue
                                        est vide (sans retirer de la liste)
                    } else {
                        System.out.println("Aucune tâche dans la file
                                        d'attente.");
                    }
                    break;
```

```
            case 3:
                if (!taskQueue.isEmpty()) {
                    System.out.println("Exécution de la tâche : " +
                            taskQueue.poll()); // supprime le premier
                                    element, pas d'indice pas de bug
                                    (pas d'exception) si la queue
                                    vide contrairement à remove
                } else {
                    System.out.println("Aucune tâche à exécuter.");
                }
                break;

            case 4:
             // Affiche toutes les tâches présentes dans la file,
             l'ordre peut ne être garanti dans l'affichage, car
             PriorityQueue est optimisé pour extraire l'élément
             avec la plus haute priorité.
                if (!taskQueue.isEmpty()) {
                    System.out.println("Toutes les tâches dans la file
                            d'attente :");
                    for (Tache t : taskQueue) {
                        System.out.println(t);
                    }
```

```
                } else {
                    System.out.println("Aucune tâche dans
                            la file d'attente.");
                }
                break;
            case 5:
                System.out.println("Au revoir !");
                scanner.close();
                return;
            default:
                System.out.println("Choix invalide, veuillez
                            réessayer.");
        }
    }
}
```

Bienvenue dans le gestionnaire de tâches !

Menu :
1. Ajouter une tâche
2. Afficher la prochaine tâche
3. Exécuter la prochaine tâche
4. Afficher toutes les tâches
5. Quitter

Choix : 1
Entrez la description de la tâche : Réviser le code
Entrez la priorité de la tâche (1 = haute, 2 = moyenne, 3 = basse) : 2
Tâche ajoutée : Task{description='Réviser le code', priority=2}

Menu :
1. Ajouter une tâche
2. Afficher la prochaine tâche
3. Exécuter la prochaine tâche
4. Afficher toutes les tâches
5. Quitter

Choix : 1
Entrez la description de la tâche : Préparer la présentation
Entrez la priorité de la tâche (1 = haute, 2 = moyenne, 3 = basse) : 1
Tâche ajoutée : Task{description='Préparer la présentation', priority=1}

Menu :
1. Ajouter une tâche
2. Afficher la prochaine tâche
3. Exécuter la prochaine tâche
4. Afficher toutes les tâches
5. Quitter

Choix : 2
Prochaine tâche : Task{description='Préparer la présentation', priority=1}