DOCUMENTATION : COLLECTIONS & ARBRES EN JAVA

Collections en Java :

- **Interfaces Principales** :
  - `Set` : Une collection non ordonnée qui ne contient pas de doublons.
  - `List` : Une collection ordonnée qui peut contenir des doublons.
  - `Map` : Une collection qui associe des clés à des valeurs, où chaque clé est unique.
  - `Queue` : Une collection qui suit un ordre spécifique pour l'ajout et la suppression d'éléments (FIFO ou LIFO).
- **Implémentations courantes** :
  - `HashSet` : Implémente l'interface `Set` en utilisant une table de hachage pour un accès rapide.
  - `ArrayList` : Implémente l'interface `List` en utilisant un tableau redimensionnable.
  - `HashMap` : Implémente l'interface `Map` en utilisant une table de hachage.
  - `ArrayDeque` : Implémente l'interface `Queue` en utilisant un tableau redimensionnable pour les opérations de type file (FIFO) et pile (LIFO).
- **Opérations Communes aux Collections**:
  - `int size()`: Retourne le nombre d'éléments dans la collection.
  - `void clear()`: Supprime tous les éléments de la collection.
  - `boolean isEmpty()`: Vérifie si la collection est vide.
- `HashSet`:
  - `boolean add(E e)`: Ajoute l'élément `e` à l'ensemble. Retourne `true` si l'élément a été ajouté (c'est-à-dire qu'il n'était pas déjà présent).
  - `boolean remove(Object o)`: Supprime l'élément `o` de l'ensemble, s'il est présent. Retourne `true` si l'élément a été supprimé.
  - `boolean contains(Object o)`: Vérifie si l'ensemble contient l'élément `o`. Retourne `true` si l'élément est présent.

```java
import java.util.HashSet;
import java.util.Set;

public class Main {
  public static void main(String[] args) {
    Set<String> set = new HashSet<>();

    // Ajout d'éléments
    set.add("pomme");
    set.add("banane");
    set.add("orange");
    System.out.println("Set après ajout: " + set);  // Affiche: [orange, pomme, banane] (l'ordre peut

    // Tentative d'ajout d'un doublon
    boolean added = set.add("pomme"); // Retourne false
    System.out.println("Ajout doublon reussi ? " + added + " Set après tentative d'ajout de doublon:

    // Vérification de la présence d'un élément
    boolean containsPomme = set.contains("pomme"); // Retourne true
      System.out.println("Le Set contient 'pomme' ? " + containsPomme); // Affiche: true

    // Suppression d'un élément
    boolean removed = set.remove("banane"); // Retourne true
     System.out.println("Suppression de 'banane' reussie ? " + removed + " Set après suppression de '

     // Vérification de la taille
     System.out.println("Taille du set: " + set.size()); // Affiche: 2

     // Vérification si le set est vide
    System.out.println("Le set est vide ? : " + set.isEmpty()); // Affiche false
  }
}
```

- `ArrayList`:
  - `boolean add(E e)`: Ajoute l'élément `e` à la fin de la liste.
  - `void add(int index, E e)`: Insère l'élément `e` à la position `index`. Lève une exception si `index` est hors limites.
  - `E get(int index)`: Retourne l'élément situé à l'index `index`.

- `int indexOf(Object o)`: Retourne l'index de la première occurrence de l'élément `o` dans la liste, ou `-1` si l'élément n'est pas présent.
- `E remove(int index)`: Supprime l'élément situé à l'index `index` et le retourne.

```
import java.util.ArrayList;
import java.util.List;

public class Main {
  public static void main(String[] args) {
    List<String> list = new ArrayList<>();

    // Ajout d'éléments
    list.add("pomme");
    list.add("banane");
    list.add("orange");
     System.out.println("Liste après ajout: " + list); // Affiche: [pomme, banane, orange]

    // Ajout à un index spécifique
    list.add(1, "kiwi");
     System.out.println("Liste après insertion de 'kiwi' à l'index 1 : " + list); // Affiche: [pomme,

    // Récupération d'un élément par index
    String element = list.get(2);
     System.out.println("Element à l'index 2: " + element); // Affiche: banane

    // Recherche de l'index d'un élément
    int index = list.indexOf("orange");
    System.out.println("L'index de 'orange' est : " + index); // Affiche: 3

    // Suppression d'un élément par index
    String removedElement = list.remove(0);
   System.out.println("Element supprimé à l'index 0 : " + removedElement + " Liste après suppression:

    // Vérification de la taille
     System.out.println("Taille de la liste: " + list.size()); // Affiche: 3

     // Vérification si la liste est vide
     System.out.println("La liste est vide ? : " + list.isEmpty()); // Affiche false
  }
}
```

- **HashMap**:
  - `V put(K key, V value)`: Ajoute ou remplace l'association de la clé `key` avec la valeur `value`. Retourne l'ancienne valeur associée à la clé, ou `null` si la clé n'existait pas.
  - `V get(Object key)`: Retourne la valeur associée à la clé `key`, ou `null` si la clé n'est pas présente.
  - `V remove(Object key)`: Supprime l'association de la clé `key` et retourne la valeur associée, ou `null` si la clé n'existait pas.
  - `boolean containsKey(Object key)`: Vérifie si la clé `key` existe dans la `Map`.
  - `boolean containsValue(Object value)`: Vérifie si la valeur `value` existe dans la `Map`.
  - `Set keySet()`: Retourne un `Set` contenant toutes les clés de la `Map`.

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

public class Main {
    public static void main(String[] args) {
        Map<String, Integer> map = new HashMap<>();

        // Ajout de paires clé-valeur
        map.put("pomme", 1);
        map.put("banane", 2);
        map.put("orange", 3);
```

```
        System.out.println("Map après ajout: " + map); // Affiche: {orange=3, banane=2, pomme=1} (l'

        // Mise à jour d'une valeur
        Integer oldValue = map.put("banane", 5);
        System.out.println("Map après mise à jour de 'banane', ancienne valeur : " + oldValue + " Map

        // Récupération d'une valeur par clé
        Integer value = map.get("orange");
        System.out.println("La valeur de la clé 'orange' est: " + value); // Affiche: 3

        // Vérification de l'existence d'une clé
        boolean keyExists = map.containsKey("pomme");
        System.out.println("La map contient la clé 'pomme' ? " + keyExists); // Affiche: true

        // Vérification de l'existence d'une valeur
        boolean valueExists = map.containsValue(5);
        System.out.println("La map contient la valeur '5' ? " + valueExists); // Affiche: true

        // Suppression d'une entrée
        Integer removedValue = map.remove("pomme");
        System.out.println("La valeur associée à la clé 'pomme' a été supprimée : " + removedValue +

        // Récupération de l'ensemble des clés
        Set<String> keys = map.keySet();
        System.out.println("Les clés de la map : " + keys); // Affiche: [orange, banane]

        // Vérification de la taille
        System.out.println("Taille de la map: " + map.size()); // Affiche: 2

        // Vérification si la map est vide
        System.out.println("La map est vide ? : " + map.isEmpty()); // Affiche false
        }
    }
```

- `ArrayDeque`:
  - **Opérations de File (FIFO):**
    - `boolean offer(E e)`: Ajoute l'élément `e` à la fin de la file.
    - `E poll()`: Supprime et retourne l'élément en tête de file. Retourne `null` si la file est vide.
  - **Opérations de Pile (LIFO):**
    - `void push(E e)`: Ajoute l'élément `e` au sommet de la pile.
    - `E pop()`: Supprime et retourne l'élément au sommet de la pile. Lève une exception si la pile est vide.
  - `boolean offerFirst(E e) / boolean offerLast(E e)` : Ajoute un element au debut ou a la fin de la liste.
  - `void addFirst(E e) / void addLast(E e)` : Ajoute un element au debut ou a la fin de la liste.
  - `E getFirst() / E getLast()` : Retourne le premier ou le dernier élément de la liste.
  - `E peekFirst() / E peekLast()`: Retourne le premier ou le dernier élément de la liste sans le supprimer.

```
import java.util.ArrayDeque;
import java.util.Queue;

public class Main {
    public static void main(String[] args) {
        ArrayDeque<Integer> queue = new ArrayDeque<>(); // Utilisation en tant que Queue (FIFO)

        // Opérations de file (FIFO)
        queue.offer(1);
        queue.offer(2);
        queue.offer(3);
        System.out.println("File après ajout: " + queue); // Affiche :

        Integer polledElement = queue.poll();
        System.out.println("Element retiré de la file (poll): " + polledElement + " File après poll :
```

```
        ArrayDeque<Integer> stack = new ArrayDeque<>(); // Utilisation en tant que Pile (LIFO)

        // Opérations de pile (LIFO)
        stack.push(1);
        stack.push(2);
        stack.push(3);
        System.out.println("Pile après ajout: " + stack); // Affiche:

        Integer poppedElement = stack.pop();
        System.out.println("Element retiré de la pile (pop): " + poppedElement + " Pile après pop: "

        ArrayDeque<Integer> deque = new ArrayDeque<>();

        deque.offerFirst(1);
        deque.offerLast(2);
        System.out.println("Deque apres offerFirst et offerLast : " + deque); // Affiche:

        deque.addFirst(0);
        deque.addLast(3);
        System.out.println("Deque apres addFirst et addLast : " + deque); // Affiche:

        int firstElement = deque.getFirst();
        int lastElement = deque.getLast();
        System.out.println("Premier element de la deque : " + firstElement + " Dernier element de l

        int peekedFirstElement = deque.peekFirst();
        int peekedLastElement = deque.peekLast();
        System.out.println("Premier element sans le retirer de la deque : " + peekedFirstElement

        // Vérification de la taille
        System.out.println("Taille de la deque: " + deque.size()); // Affiche: 4

        // Vérification si la deque est vide
        System.out.println("La deque est vide ? : " + deque.isEmpty()); // Affiche false
        }
    }
```

Parcours de Collections :

- **Boucle `for`** : Utilisée pour itérer sur les éléments d'une collection, mais sans possibilité de modifier la collection en cours de parcours.

```
import java.util.ArrayList;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("pomme");
        list.add("banane");
        list.add("orange");

        for (String fruit : list) {
            System.out.println(fruit);
        }
       // Affiche : pomme, banane, orange
    }
}
```

- **`Iterator`** : Utilisé pour parcourir et modifier une collection. Permet la suppression d'éléments pendant l'itération.
  - `Iterator`:

  * `hasNext()`: Retourne `true` s'il y a encore des éléments à parcourir.
  * `next()`: Retourne l'élément suivant dans la collection.
  * `remove()`: Supprime le dernier élément retourné par `next()`.

```java
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("pomme");
        list.add("banane");
        list.add("orange");

        Iterator<String> iterator = list.iterator();
        while (iterator.hasNext()) {
            String fruit = iterator.next();
            System.out.println(fruit);
            if (fruit.equals("banane")) {
                iterator.remove(); // Suppression de "banane"
            }
        }
         System.out.println("Liste après modification : " + list); // Affiche : pomme, orange
    }
}
```

- `ListIterator` (pour `List` uniquement):
  * Hérite de `Iterator` et ajoute des méthodes pour parcourir la liste dans les deux sens et modifier la liste pendant le parcours.
  * `previous()`: Retourne l'élément précédent.
  * `add(E e)`: Ajoute un élément à la position actuelle de l'itérateur.

```java
import java.util.ArrayList;
import java.util.List;
import java.util.ListIterator;

public class Main {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("pomme");
        list.add("banane");
        list.add("orange");

        ListIterator<String> listIterator = list.listIterator();
        while (listIterator.hasNext()) {
          String fruit = listIterator.next();
            if (fruit.equals("banane")) {
                 listIterator.add("kiwi"); // Ajout de "kiwi" après "banane"
            }
            System.out.println(fruit);
        }
           System.out.println("Liste après modification : " + list); // Affiche: pomme, banane, kiwi,

        while (listIterator.hasPrevious()) {
           String fruit = listIterator.previous();
           System.out.println("Element précedent : " + fruit);
       }
        // Affiche : orange, kiwi, banane, pomme
    }
}
```

Arbres en Java :

- **Parcours en profondeur d'abord (DFS):**
  - Approche récursive pour explorer chaque branche de l'arbre avant de passer à la suivante.
- **Parcours en largeur d'abord (BFS):**
  - Approche itérative utilisant une file d'attente pour explorer l'arbre niveau par niveau.
- **Arbre binaire ordonné:**
  - Chaque nœud a au maximum deux enfants, un à gauche et un à droite, avec la propriété que les valeurs du sous-arbre gauche sont inférieures ou égales à la valeur du nœud et celles du sous-arbre droit sont supérieures.
  - **Méthodes:**
    * `insert(int value)`: Insère une nouvelle valeur dans l'arbre, en respectant l'ordre.
    * `search(int value)`: Recherche une valeur spécifique dans l'arbre et retourne le nœud correspondant ou `null` si la valeur n'est pas trouvée.

```
class Node {
    int value;
    Node left;
    Node right;

    public Node(int value) {
        this.value = value;
        this.left = null;
        this.right = null;
    }
}

class BinarySearchTree {
    Node root;

    public void insert(int value) {
        root = insertRecur(root, value);
    }

     private Node insertRecur(Node n, int value) {
        if (n == null) {
            return new Node(value);
        }

        if (value <= n.value) {
          n.left = insertRecur(n.left, value);
        } else {
            n.right = insertRecur(n.right, value);
        }
         return n;
    }
      public Node search(int value) {
          Node n = root;
        while ((n != null) && (n.value != value)) {
          if (value <= n.value) {
            n = n.left;
         } else {
              n = n.right;
         }
        }
        return n;
   }
}


public class Main {
    public static void main(String[] args) {
        BinarySearchTree bst = new BinarySearchTree();
        bst.insert(5);
```

```
        bst.insert(3);
        bst.insert(7);
        bst.insert(1);
        bst.insert(4);
        bst.insert(6);
        bst.insert(8);


        Node found = bst.search(4);
        if(found != null) {
           System.out.println("Le noeud est trouvé :" + found.value); // Affiche: Le noeud est trouvé
       } else {
         System.out.println("Le noeud n'est pas trouvé.");
        }
         Node notFound = bst.search(10);
          if(notFound != null) {
             System.out.println("Le noeud est trouvé : " + notFound.value);
          } else {
             System.out.println("Le noeud n'est pas trouvé."); // Affiche: Le noeud n'est pas trouvé.
          }

    }
}
```