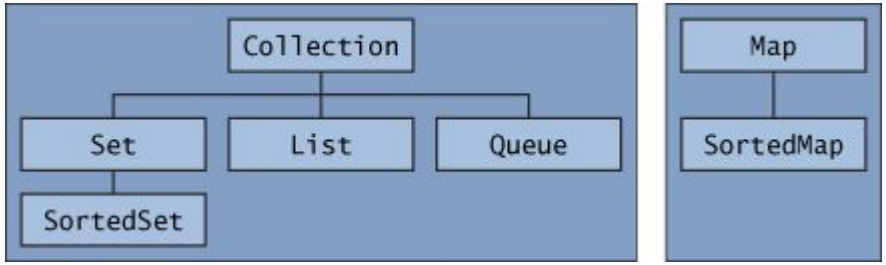
Collections

Collections

- Les collections permettent de gérer des groupes d'objets. Elles remplacent les tableaux classiques quand il faut manipuler des données dynamiques.

Collections



a. Interface Collection

- Interface racine pour toutes les collections.
- Elle est directement étendue par :
  - **List** : Collection ordonnée permettant les doublons.
  - **Set** : Collection non ordonnée qui ne permet pas les doublons.
  - **Queue** : Collection traitée en fonction de l'ordre d'insertion (FIFO).

b. Interface Map

- Une Map gère des paires clé-valeur où les clés sont uniques.

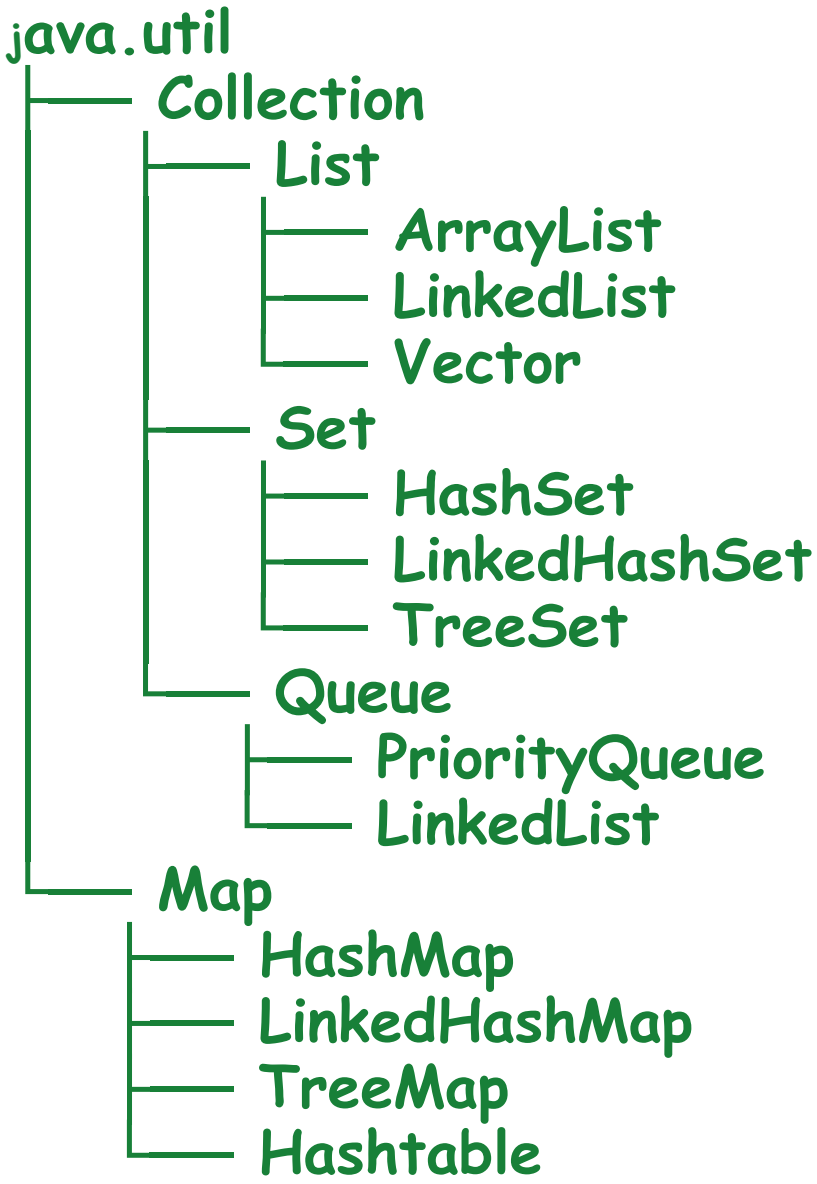
Collections

Interface Collection

C'est la racine de presque toutes les interfaces des collections. Elle définit des méthodes générales pour manipuler des groupes d'objets :

- Ajout d'éléments : `add(E e)`, `addAll(Collection<? extends E> c)`.
- Suppression d'éléments : `remove(Object o)`, `clear()`.
- Vérifications : `isEmpty()`, `contains(Object o)`.
- Taille : `size()`.

Les collections sont des interfaces génériques



Les Listes (`List`)

Une **liste** maintient l'ordre d'insertion et permet des doublons.

Méthodes principales :

- **Accès aux éléments par index :** `get(int index)`, `set(int index, E element)`.
- **Ajout :** `add(E e)`, `add(int index, E element)`.
- **Recherche :** `indexOf(Object o)`, `lastIndexOf(Object o)`.

Les Listes (`List`)

Implémentations courantes :

- **`ArrayList`** : Tableau dynamique (rapide pour accéder aux éléments).
- **`LinkedList`** : Liste chaînée (efficace pour ajouter/supprimer).

Exemple :

```
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList<String> fruits = new ArrayList<>();
        fruits.add("Pomme");
        fruits.add("Banane");
        fruits.add("Orange");

        System.out.println("Liste des fruits : " + fruits);

        fruits.remove("Banane");
        System.out.println("Après suppression : " + fruits);

        System.out.println("Premier fruit : " + fruits.get(0));
    }
}
```

Les Ensembles (`Set`)

Un **ensemble** ne permet pas de doublons.

Implémentations courantes :

- `HashSet` : Basé sur une table de hachage (non ordonné).
- `TreeSet` : Ordonné naturellement ou selon un comparateur.

Exemple :

```
import java.util.HashSet;

public class Main {
    public static void main(String[] args) {
        HashSet<Integer> nombres = new HashSet<>();
        nombres.add(10);
        nombres.add(20);
        nombres.add(10); // Ignoré, car doublon

        System.out.println("Ensemble de nombres : " + nombres);
    }
}
```

Exemple :

```
// Java program to demonstrate the
// Sorted Set
import java.util.*;

class SortedSetExample{

    public static void main(String[] args)
    {
        SortedSet<String> ts  = new TreeSet<String>();

        // Adding elements into the TreeSet
        // using add()
        ts.add("India");
        ts.add("Australia");
        ts.add("South Africa");

        // Adding the duplicate
        // element
        ts.add("India");

        // Displaying the TreeSet
        System.out.println(ts);

        ts.remove("Australia");
        System.out.println("Set after removing "+ "Australia:" + ts);
    }
}
```

Queue (Queue)

Une **Queue** (file d'attente) est une structure de données qui suit le principe **FIFO** (*First In, First Out*), où le premier élément ajouté est le premier à être retiré.

```
import java.util.LinkedList;
import java.util.Queue;

public class Main {
    public static void main(String[] args) {
        // Création d'une file d'attente
        Queue<String> fileAttente = new LinkedList<>();

        // Ajout de clients dans la file
        fileAttente.add("Alice");
        fileAttente.add("Bob");
        fileAttente.add("Charlie");

        System.out.println("File d'attente initiale : " + fileAttente);

        // Servir les clients un par un
        while (!fileAttente.isEmpty()) {
            String clientServi = fileAttente.poll(); // Récupère et supprime le premier élément
            System.out.println("Client servi : " + clientServi);
            System.out.println("File d'attente mise à jour : " + fileAttente);
        }
    }
}
```

Les Maps (`Map`)

Une **map** stocke des paires clé-valeur. Les clés sont uniques.

Méthodes principales :

- **Ajout et mise à jour :** `put(K key, V value)`.
- **Accès :** `get(Object key)`, `containsKey(Object key)`, `containsValue(Object value)`.
- **Suppression :** `remove(Object key)`.
- **Entrées :** `entrySet()`, `keySet()`, `values()`.

Les Maps (`Map`)

Implémentations courantes :

- `HashMap` : Rapide, non ordonné.
- `TreeMap` : Ordonné selon les clés.

Exemple :

```
import java.util.HashMap;

public class Main {
    public static void main(String[] args) {
        HashMap<String, Integer> notes = new HashMap<>();
        notes.put("Mathématiques", 18);
        notes.put("Physique", 15);
        notes.put("Mathématiques", 19); // Remplace la valeur existante

        System.out.println("Notes : " + notes);
    }
}
```

Classes utilitaires : `Collections` et `Arrays`

1- Tri :

```
ArrayList<Integer> list = new ArrayList<>(Arrays.asList(5, 1, 3));
Collections.sort(list);
System.out.println(list); // [1, 3, 5]
```

2- Recherche binaire :

```
int index = Collections.binarySearch(list, 3);
System.out.println(index); // 1
```

3- Transformation tableau <-> liste :

```
List<String> list = Arrays.asList("Alice", "Bob");
```

Exercice 1: Ajouter des étudiants dans une liste

Projet : Gestion des étudiants

1. Description du projet

Vous devez créer une application Java permettant de gérer les informations d'étudiants. Chaque étudiant a un `id`, un `nom` et une `note`. L'application doit permettre d'effectuer plusieurs opérations sur une liste d'étudiants :

- Ajouter un étudiant.
- Supprimer un étudiant par `id`.
- Rechercher un étudiant par `id`.
- Trier les étudiants par `nom`.
- Calculer la moyenne des notes des étudiants.
- Vérifier si un étudiant existe déjà (uniquement par `id`).
- Afficher la liste de tous les étudiants.

2. Classes et Structure

1. **Classe `Etudiant`** : Représente les informations d'un étudiant.
2. **Classe `GestionEtudiants`** : Gère toutes les opérations sur les étudiants.
3. **Collections utilisées** :
    - **`List` (ArrayList)** pour stocker les étudiants et effectuer des opérations comme l'ajout, la recherche, et le tri.
    - **`Map` (HashMap)** pour lier l'`id` de l'étudiant à l'objet `Etudiant` (recherche rapide par `id`).
    - **`Set` (HashSet)** pour vérifier l'unicité des `id`.