ECOLE MAROCAINE DES SCIENCES DE L'INGENIEUR
Membre de
HONORIS UNITED UNIVERSITIES

COLLECTIONS EN JAVA

3IIR - POO2

LES COLLECTIONS EN JAVA ?

- Les collections en Java représentent les structures de stockage standards (listes, ensembles, dictionnaires).
- Elles offrent une interface unifiée pour insérer, supprimer, trier, rechercher ou parcourir les éléments sans avoir à réimplémenter les mécanismes de base.

QU'EST-CE QU'UNE COLLECTION?

- **Une collection est un objet qui permet de stocker un ensemble d'objets.**
  - Permet de manipuler des groupes d'objets de façon homogène, quel que soit le type.
  - Les objets contenus dans une collection peuvent être de même type ou de types différents (mais **compatibles**).
- **Une collection prend en charge les opérations courantes comme l'ajout, la suppression, la recherche, le tri, …etc.**

L'API COLLECTIONS

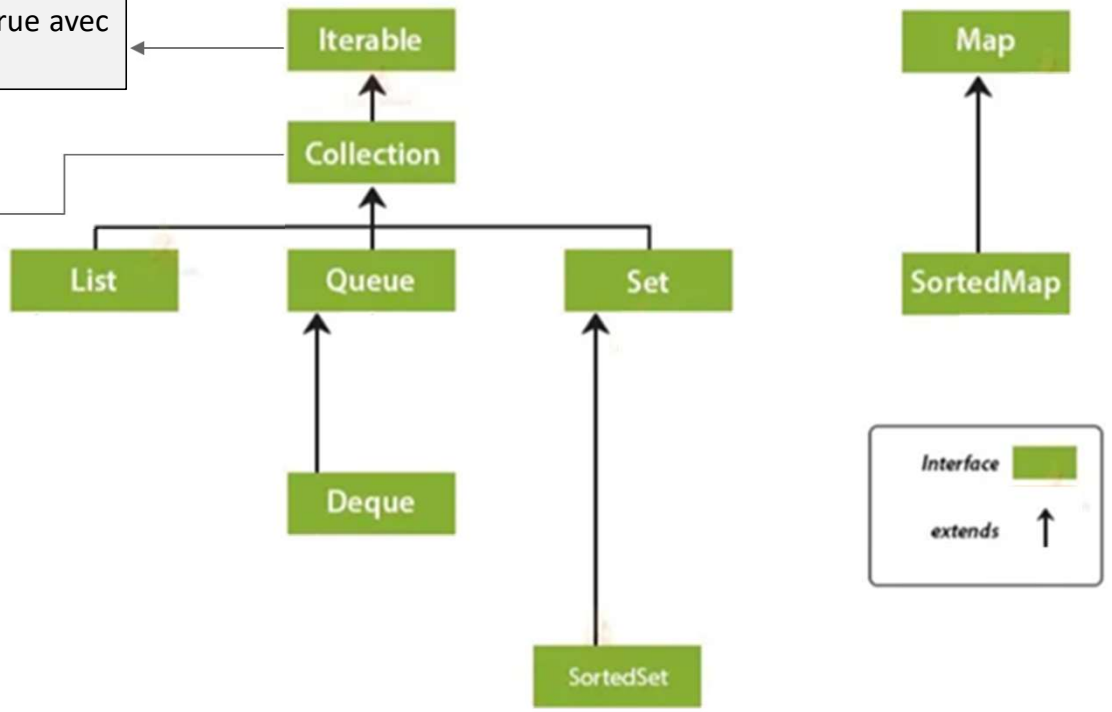
- En Java, les collections font partie de la ***Java Collections Framework.***
  - Située dans le package **java.util** => ***import java.util.****
  - C'est **l'API** officielle de Java pour gérer les **collections** d'objets.
  - **Composée d'un ensemble d'interfaces et de classes** qui les implémentent.
    - Faciles à utiliser : syntaxe claire et cohérente.
    - Permettent de produire du code fiable et performant.
    - Offrent une grande souplesse et flexibilité, dans la gestion des groupes d'objets, par rapport aux tableaux classiques.

L'API COLLECTIONS : INTERFACES

- **List**, **Queue**, **Set et Map** sont les principales **interfaces** de **l'API Collections**.

Donne la capacité à une collection d'être parcourue avec une boucle **forEach.**

Décrit les capacités génériques des collections, telles que l'ajout et la suppression d'éléments.



Remarquez que Map n'est pas une sous interface de **Collection**, ***car ce n'est pas une Collection au sens strict.***

- ***Les éléments d'une map sont des paires clé/valeur qui nécessitent des méthodes spécifiques.***

L'API COLLECTIONS : INTERFACES

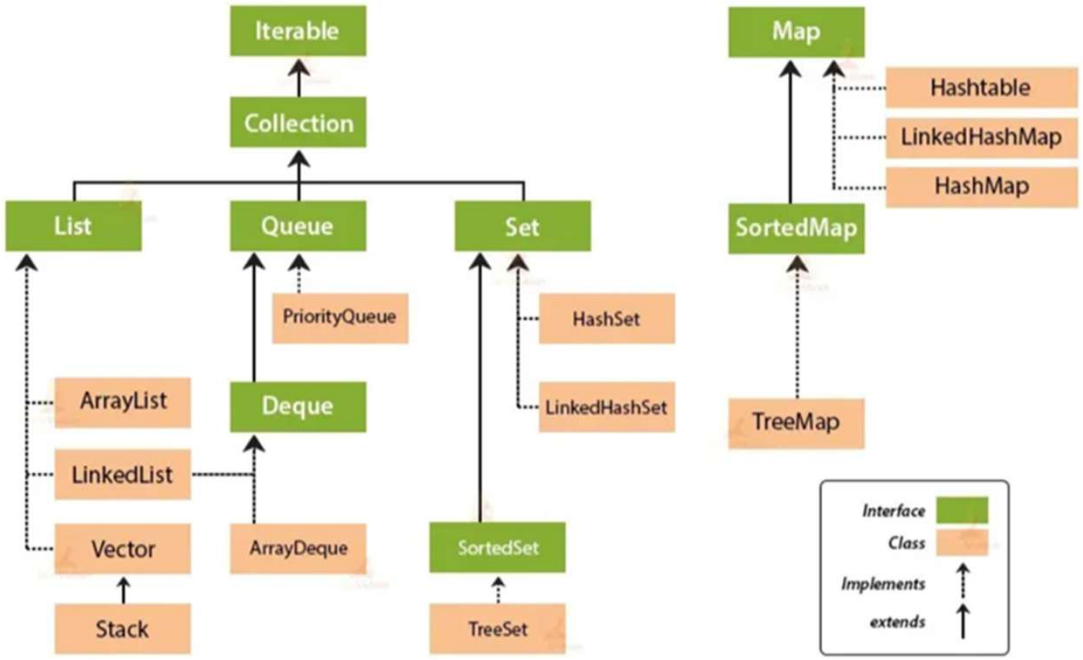
- **List : liste ordonnée** d'éléments qui sont accessibles directement via un indice entier.
- **Queue : file d'attente** qui ordonne ses éléments selon un ordre FIFO, mais pouvant être LIFO selon l'implémentation utilisée.
- **Set : ensemble** d'éléments qui n'autorise pas les doublons.
- **Map : table de correspondance** qui associe des clés à des valeurs, sans autoriser les clés dupliquées. Chaque élément est une *paire clé/valeur.*

L'API COLLECTIONS : INTERFACES

- **Les interfaces sont utilisées pour des raisons de flexibilité.**
  - En programmant avec des interfaces plutôt qu'avec des classes concrètes, un programme n'est pas lié à une implémentation spécifique.
  - Cela permet de remplacer facilement une classe par une autre plus performante, à condition qu'elle implémente la même interface.
    - Ainsi, on peut par exemple remplacer un **ArrayList** par un **LinkedList** sans modifier le reste du code, car les deux classes implémentent l'interface **List**.

L'API COLLECTIONS : CLASSES

- Chaque interface de l'API est implémentée par **un ensemble de classes,** chacune offrant des fonctionnalités spécifiques selon les besoins.

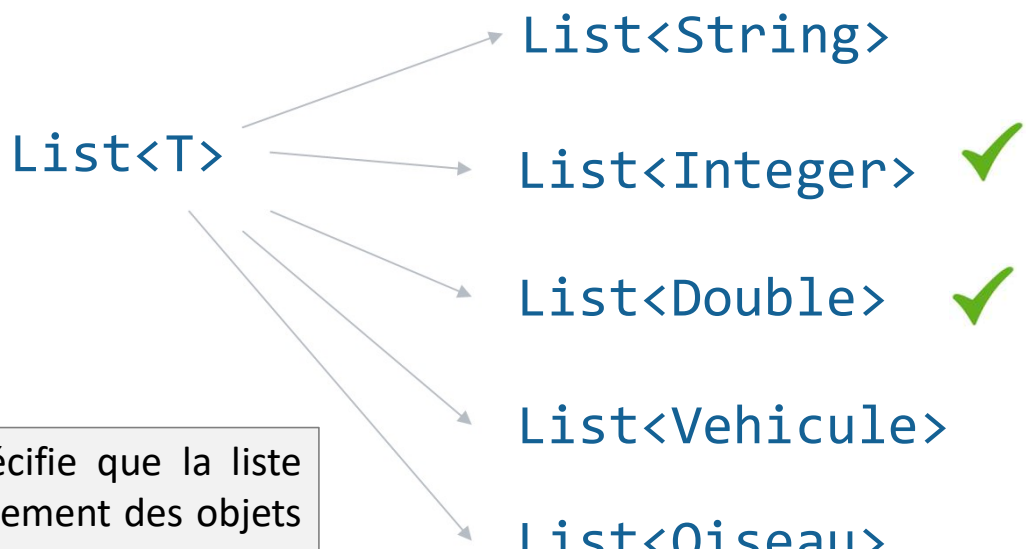


On peut voire l'API Collections comme une boîte à outils :

- Chaque interface est une **spécification abstraite**, et chaque classe est **un outil spécialisé** qui répond à un type de besoin précis.
- L'objectif est de **choisir l'outil adapté sans changer la structure du programme**, grâce à la puissance des interfaces.

L'API COLLECTIONS : TYPE GÉNÉRIQUE

- Les collections en Java sont **génériques => List<T>, set<T>, map<K,V> , …**
  - Cela veut dire qu'elles ont été conçues avec un **type paramétré (***souvent représenté par la lettre* **T),** laissant ainsi à l'utilisateur la possibilité de choisir le type des éléments que les collections vont contenir.



List<int> ⊗

List<double> ⊗

List<String> spécifie que la liste va contenir seulement des objets de type **String**.

Les collections en Java ne peuvent stocker que des **objets** pas des types primitifs**. Utilisez les classes wrapper** comme Integer, Double, Character, etc. qui permettent de traiter les types primitifs comme des objets.

- Cela garantit que seuls les objets du type spécifié peuvent être ajoutés à la collection (type safety).

L'INTERFACE COLLECTION

- L'interface **Collection** est l'interface racine de toutes les collections. Elle définit plusieurs opérations de base pour manipuler les éléments d'une collection.
  - **boolean add(E element)** : ajoute un élément et retourne si l'opération a réussi ou non.
  - **boolean remove(Object obj):** supprime un élément et retourne si l'opération a réussi ou non (remove utilise equals).
  - **boolean isEmpty()** : retourne true si la collection est vide.
  - **int size()** : retourne le nombre d'éléments.
  - **void clear():** vide la collection.
  - **boolean contains(Object obj)** : retourne true si l'élément existe (contains utilise equals).
  - **boolean equals(Object obj)** : retourne true si les deux collections sont égales.
- Les interfaces **List**, **Queue** et **Set** étendent ces opérations de base en ajoutant leurs propres méthodes spécifiques.

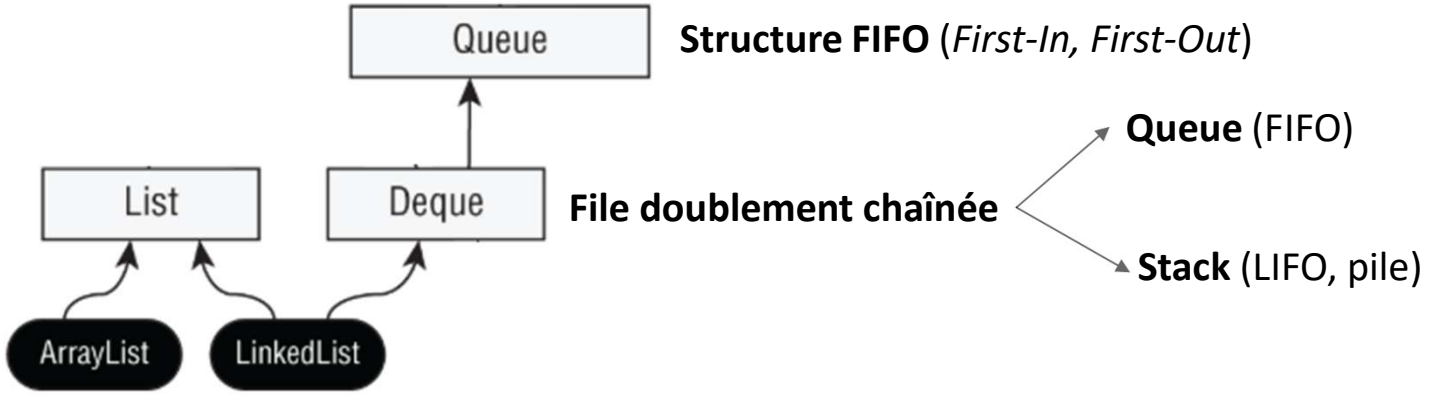
COLLECTIONS : LIST

L'INTERFACE LIST

- **List<T>** : Collection ordonnée d'éléments (ordre d'insertion), qui **accepte les doublons** et qui **permet l'accès direct** aux éléments via un index.
  - Fournit des méthodes qui servent **à manipuler les éléments via les index.**
    - **void add (int index, E element)** : Ajoute un élément à l'index indiqué et décale les autres éléments vers la fin.
    - **E get(int index)** : Renvoie l'élément correspondant à l'index indiqué.
    - **E set(int index, E e) :** Remplace l'élément à l'index indiqué et renvoie l'élément original. Lève une exception IndexOutOfBoundsException si l'index est invalide.
    - **int indexOf(Object o)** : Renvoie l'index du premier élément correspondant ou -1 s'il n'est pas trouvé.
    - **E remove(int index)** : Supprime l'élément correspondant à l'index indiqué et décale les autres vers le début.

LIST : ARRAYLIST VS LINKEDLIST

- ***Les principales classes qui implémentent l'interface, List, sont* ArrayList et LinkedList.**



Implémentation	Stockage des éléments	Caractéristiques	Quand l'utiliser ?
`ArrayList<T>`	**Tableau dynamique**	Accès direct rapide (O(1)) mais coûteux en ajout et suppression (O(n))	Liste avec beaucoup de lectures, ou autant de lectures que d'écritures.
`LinkedList<T>`	**Liste doublement chaînée**	Insertion/suppression rapide (O(1)) mais accès lent (O(n))	Si vous modifiez souvent la liste (ajouts, suppressions).

LIST : ARRAYLIST VS LINKEDLIST

Constructeurs de ArrayList et LinkedList

```
public static void main(String[] args) {
 List<Integer> premiers = new ArrayList<Integer>();
 List<String>  voitures = new ArrayList<>();
 List<> tab = new ArrayList<Double>();
}
```

Constructeur sans arguments, créant un tableau avec une capacité initiale de 10 éléments.

Constructeur sans arguments, créant une liste chainée vide.

```
public static void main(String[] args) {
    List<String> voitures = new LinkedList<>();
}
```

LIST : ARRAYLIST VS LINKEDLIST

Constructeurs de ArrayList et LinkedList

- **Constructeur avec un argument précisant la capacité initiale**

LinkedList n'a pas de constructeur qui prend un entier en argument.

```
public static void main(String[] args) {

 List<Integer>  premiers = new ArrayList<>(5);

}
```

```
public static void main(String[] args) {

 List<Integer> premiers = new LinkedList<>(5);

}
```

- **Constructeur ayant comme argument toute autre collection (pour initialiser la liste avec des valeurs initiales)**

```
public static void main(String[] args) {

 List<Integer>  premiers = new ArrayList<>(List.of(2,3,5,7,11));
 premiers.add(13);
 List<Integer>  nombres = new LinkedList<>(premiers);

}
```

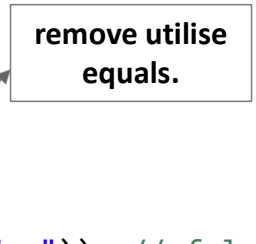
- **Crée une liste non modifiable**

```
List<Integer>  uneListe = List.of(1,2,3);
uneListe.add(4);
```

liste non modifiable

LIST : ARRAYLIST VS LINKEDLIST

```java
public static void main(String[] args) {

    List<String> voitures = new ArrayList<>(); //new LinkedList<>();

     System.out.println(voitures.isEmpty()); // true
     System.out.println(voitures.size()); // 0

     //Ajouter des voitures
     voitures.add("Toyota");
     voitures.add("BMW");
     voitures.add("Audi");

     //Afficher avec foreach
     for(String v : voitures)
       System.out.println(v);

     //Supprimer Mercedes des voitures
     System.out.println(voitures.remove("Mercedes")); // false

     System.out.println(voitures.size()); // 3
     voitures.clear(); //vide la collection

}
```



LIST : ARRAYLIST VS LINKEDLIST

- **Il est recommandé de déclarer une collection en utilisant l'interface** (comme **List**) plutôt que la classe d'implémentation (comme **ArrayList** ou **LinkedList**)

- **En utilisant les interfaces :**
  - **On ne dépend que du contrat** (l'interface), ce qui rend le code plus souple.
  - **On peut changer l'implémentation** (ex. passer de **ArrayList** à **LinkedList**) sans modifier le reste du code, tant qu'on respecte l'interface.
  - **On peut traiter différents types de collections** de la même manière, en tirant parti du polymorphisme.

```
public static void main(String[] args) {

    List<String> voitures = new ArrayList<>();
    voitures.add("Toyota");
    voitures.add("BMW");
    voitures.add("Audi");

    System.out.println(voitures.get(0)); //Toyota
    voitures.set(0, "Jeep"); //remplace Toyota par Jeep
    System.out.println(voitures.contains("Jeep")); // true

    // Affichage avec forEach

    voitures.forEach(v -> System.out.println(v));

    // Jeep, BMW, Audi,
}
```

new LinkedList<>();

contains utilise equals.

LIST : ARRAYLIST VS LINKEDLIST

- Il est fortement recommandé d'utiliser les **collections génériques** en Java.

```
public static void main(String[] args) {

List numbers = new ArrayList(List.of(1,2,3));
Integer element = (Integer)numbers.get(0); // cast obligatoire
numbers.add("une chaine!"); // types incompatibles autorisés
}
```

⚠ Warning

```
public static void main(String[] args) {

List<Integer> numbers = new ArrayList<>(List.of(1,2,3));
Integer element = numbers.get(0); // aucun cast requis
numbers.add("une chaine!"); // ne compile pas
}
```

- **En utilisant les collections génériques :**
  - On spécifie le type des éléments qu'une collection peut contenir.
  - On évite que des éléments de type non compatible soient ajoutés à la collection (code plus sûr)
  - On évite les conversions de types (casts) inutiles.

COLLECTIONS : SET

L'INTERFACE SET

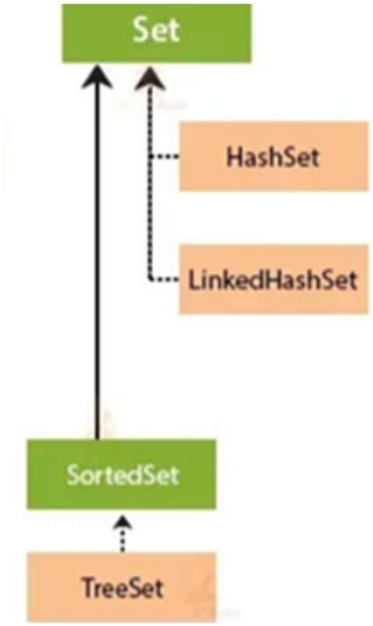
- **Set<T> : Correspond à la définition mathématique d'un ensemble => collection d'éléments sans doublons.**

  - Vous pouvez créer un Set non modifiable avec la méthode statique ***of***

```
Set<Character> letters = Set.of('c', 'a', 't');
```

  - Vous pouvez faire une copie non modifiable d'un Set existant avec la méthode ***copyOf***

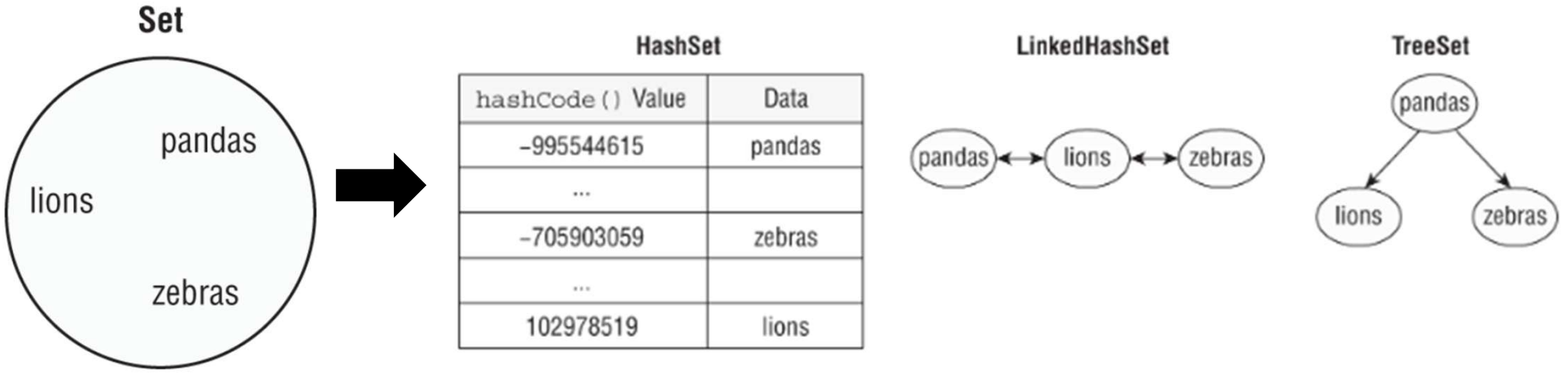
```
Set<Character> setCopie = Set.copyOf(letters);
```



- **Set n'a pas de méthodes spécifiques à part celles héritées de Collection.**
- **Set est implémentée par les classes HashSet, LinkedHashSet et TreeSet**

IMPLÉMENTATIONS DU SET

Implémentation	Stockage des éléments	Caractéristiques	Quand l'utiliser ?
HashSet<T>	**Table de hachage**	Accès très rapide, ordre non garanti	Stocker des éléments uniques sans souci d'ordre
TreeSet<T>	**Arbre binaire**	Trié en ordre naturel, plus lent que le HashSet	Garder toujours l'ensemble trié
LinkedHashSet<T>	**Table de hachage + liste chaînée**	Conserve l'ordre d'insertion, légèrement plus lent que le HashSet	Mélanger entre HashSet et List



IMPLÉMENTATIONS DU SET

HashSet => Ordre aléatoire

```
Set<Integer> set = new HashSet<>();
boolean b1 = set.add(66); // true
boolean b2 = set.add(10); // true
boolean b3 = set.add(66); // false (doublon)
boolean b4 = set.add(8); // true
//Affichage du set
for (Integer value: set)
System.out.print(value + ','); // 66,8,10,
```

Un set stocke des éléments uniques!

TreeSet => Trié en ordre naturel

```
Set<Integer> set = new TreeSet<>();
set.add(66);
set.add(10);
set.add(8);
set.add(66); //doublon détecté
//Affichage du set
set.forEach(System.out::println);// 8,10,66
```

LinkedHashSet => Ordre d'insertion

```
Set<Integer> set = new LinkedHashSet<>();
set.add(8);
set.add(66);
set.add(10);
set.add(66); //doublon détecté
// Affichage du set
set.forEach(e -> System.out.println(e));// 8,66,10
```

IMPLÉMENTATIONS DU SET

```java
import java.util.Objects;
public class Personne {
     private String nom;
     private int age;

     public Personne(String nom, int age) {
     this.nom = nom;
     this.age = age;
     }
     @Override
     public boolean equals(Object o) {
          if (this == o) return true;
          if (!(o instanceof Personne)) return false;
          Personne p = (Personne) o;
          return age == p.age && nom.equals(p.nom);
     }
     @Override
     public int hashCode() {
     return Objects.hash(nom, age);
     }
}
```

Deux personnes sont identiques si elles ont même nom et même âge

Un set stocke des éléments uniques!

```java
Set<Personne> set = new HashSet<>();
set.add(new Personne("Alia", 25));
```

- Pour le **HashSet**, la méthode **add** utilise le **hashCode** et la méthode **equals** pour vérifier l'existence d'un élément.
- ***Il faut donc redéfinir ces deux méthodes dans la classe Personne.***
  - **`hashCode()`** : Sert à localiser rapidement l'élément dans la table de Hachage.
  - **`equals(Object O)`** : Sert à comparer réellement l'égalité entre les deux éléments.

IMPLÉMENTATIONS DU SET

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

public class TestHashSet {

public static void main(String[] args) {
    Set<Personne> set = new HashSet<>();
    set.add(new Personne("Alia", 25));
    set.add(new Personne("Alia", 25)); //ignoré, car même nom et même age
    set.add(new Personne("Alia", 21)); //Accepté, car âge différent
    for(Personne p : set)
      System.out.println(p);
  }

}
```

```
[nom=Alia, age=21]
[nom=Alia, age=25]
```

HashSet => Ordre aléatoire

IMPLÉMENTATIONS DU SET

Lorsqu'une classe **implémente l'interface Comparable**, ses objets deviennent **comparables**.

L'interface **Comparable<T>** est utilisée en Java pour définir un ordre naturel entre les objets d'une même classe.

```
public class Personne implements Comparable<Personne> {

private String nom;
private int age;
…

@Override
public int compareTo(Personne autre) {
     int result = this.nom.compareTo(autre.nom);
      if (result == 0) {
         return Integer.compare(this.age, autre.age);
        }
      return result;
     }
}
```

Le tri sera automatiquement effectué par nom puis par âge, car c'est ce que définit compareTo.

```
Set<Personne> set = new TreeSet<>(); // ordre naturel
set.add(new Personne("Alia", 25));
```

- Un **TreeSet** effectue **automatiquement** le tri **lors de l'insertion des éléments.** Il **compare** donc les éléments pour définir l'ordre dans l'ensemble.
- Pour que deux personnes soient comparables, la classe **Personne** doit implémenter l'interface **Comparable** et redéfinir la méthode **compareTo.**
- La méthode **add** utilisera la méthode **compareTo pour comparer les personnes.**

IMPLÉMENTATIONS DU SET

```
import java.util.Set;
import java.util.TreeSet;

public class TestTreeSet {

public static void main(String[] args) {
    Set<Personne> set = new TreeSet<>(); // ordre naturel
    set.add(new Personne("Alia", 25));
    set.add(new Personne("Alia", 25)); //doublon détecté
    set.add(new Personne("Alia", 21)); //Accepté
    set.forEach(p -> System.out.println(p));
    }

}
```

```
[nom=Alia, age=21]
[nom=Alia, age=25]
```

Un **TreeSet** trie s**es éléments !**

La méthode **CompareTo** est utilisée à la fois pour déterminer si deux personnes sont considérées identiques et pour définir l'ordre naturel dans lequel **les personnes seront triées lors de leur insertion.**

Si deux personnes ont même nom, elles sont triées par âge.

IMPLÉMENTATIONS DU SET

- Pour que le **`TreeSet` utilise un ordre différent de l'ordre naturel :**

```
Set<Personne> set = new TreeSet<>(……);
```

Vous devez créer un **objet `Comparator`** qui définit **un autre critère de comparaison** et le passer en argument au **TreeSet**

- Le **`TreeSet`** utilisera automatiquement l'ordre personnalisé défini par ce comparateur et lors de l'insertion, la méthode **`add()`** utilisera la méthode **`compare`** du comparateur (et non `compareTo`) pour comparer les éléments.

IMPLÉMENTATIONS DU SET

- On utilise l'interface **Comparator<T>** pour **créer des comparateurs**.
  - Le type paramètre **T** sera remplacé par le **type des objets à comparer.**
  - L'interface impose la redéfinition de la méthode **compare(T o1, T o2)** qui permet de comparer deux objets selon un critère spécifique.

Type des objets à comparer

```
import java.util.Comparator;
//Comparator pour trier les personnes par nom
class ComparateurParNom implements Comparator<Personne> {
    @Override
    public int compare(Personne p1, Personne p2) {
      return p1.getNom().compareTo(p2.getNom());
    }
}
```

IMPLÉMENTATIONS DU SET

```
Set<Personne> set = new TreeSet<>(new ComparateurParNom());
```

```
public class Personne {
     private String nom;
     private int age;

     …
      public String getNom() {
        return nom;
      }
     public int getAge() {
        return age;
     }
}
```

On passe le comparateur au TreeSet, qui trie alors les éléments selon le critère définit par la méthode compare.

```
set.add(new Personne("Alia", 25));
set.add(new Personne("Ahmed", 25));
set.forEach(p -> System.out.println(p));
```

```
[nom=Ahmed, age=25]
[nom=Alia, age=25]
```

Tri par nom

IMPLÉMENTATIONS DU SET

- La méthode statique **`Comparator.comparing(...)`** permet aussi de créer un comparateur qui indique un critère de tri basé sur un attribut spécifique d'un objet.

```
Set<Personne> set = new TreeSet<>(Comparator.comparing(Personne::getNom));
```

comparaison par nom

```
public class Personne {
     private String nom;
     private int age;

     …
      public String getNom() {
        return nom;
      }
     public int getAge() {
         return age;
     }
}
```

```
set.add(new Personne("Alia", 25));
set.add(new Personne("Alia", 25)); //doublon ignoré
set.add(new Personne("Alia", 21)); //doublon ignore
set.forEach(p -> System.out.println(p));
```

```
[nom=Alia, age=25]
```

IMPLÉMENTATIONS DU SET

```
Set<Personne> set = new TreeSet<>(Comparator.comparing(Personne::getAge));
```

Comparaison par âge

```
public class Personne {
     private String nom;
     private int age;

     …
      public String getNom() {
        return nom;
      }
     public int getAge() {
        return age;
     }
}
```

```
set.add(new Personne("Alia", 25));
set.add(new Personne("Ahmed", 25)); //ignore, même âge
set.add(new Personne("Alia", 21)); //Accepté
set.forEach(p -> System.out.println(p));
```

```
[nom=Alia, age=21]
[nom=Alia, age=25]
```

IMPLÉMENTATIONS DU SET

On peut inverser l'ordre du comparateur avec la méthode reversed()

```
Set<Personne> set = new TreeSet<>(Comparator.comparing(Personne::getNom).reversed());
```

```
public class Personne {
     private String nom;
     private int age;

     public Personne(String nom, int age) {
     this.nom = nom;
     this.age = age;
     }
      public String getNom() {
        return nom;
      }
     public int getAge() {
        return age;
     }
}
```

```
set.add(new Personne("Alia", 25));
set.add(new Personne("Ahmed", 25));
set.add(new Personne("Naser", 21));
set.forEach(p -> System.out.println(p));
```

```
[nom=Naser, age=21]
[nom=Alia, age=25]
[nom=Ahmed, age=25]
```

IMPLÉMENTATIONS DU SET

- On peut ajouter un second critère de tri en cas d'égalité sur le premier à l'aide de la méthode `thenComparing(...)`

```
Set<Personne> set = new TreeSet<>(Comparator.comparing(Personne::getNom).thenComparing(Personne::getAge));
```

S'ils ont même nom, on compare l'âge

```
public class Personne {
     private String nom;
     private int age;

     …
      public String getNom() {
        return nom;
      }
     public int getAge() {
        return age;
     }
}
```

```
set.add(new Personne("Alia", 25));
set.add(new Personne("Ahmed", 25));
set.add(new Personne("Alia", 21));
set.forEach(p -> System.out.println(p));
```

```
[nom=Ahmed, age=25]
[nom=Alia, age=21]
[nom=Alia, age=25]
```

IMPLÉMENTATIONS DU SET

- Les instances de **HashSet** peuvent ***être créées*** en utilisant :
  1. Un constructeur sans argument, créant un ensemble vide avec une capacité initiale de 16 éléments.
  2. Un constructeur avec une capacité initiale spécifique.
  3. Un constructeur avec une capacité initiale spécifique et un facteur de charge (par défaut 0,75).
  4. Toute autre collection (par exemple, une List) pour initialiser cet ensemble avec des valeurs initiales.

1
```
Set<Integer> set = new HashSet<>();
```

2
```
Set<Integer> set1 = new HashSet<>(50);
```

3
```
Set<Integer> set2 = new HashSet<>(50, 0.80f);
```

4
```
Set<Integer> set3 = new HashSet<>(List.of(1,2,3));
```

IMPLÉMENTATIONS DU SET

- Les instances de **TreeSet** peuvent ***être créées*** en utilisant :
  1. Un constructeur sans argument, créant un ensemble vide trié selon l'ordre naturel des éléments (les éléments doivent implémenter l'interface Comparable).
  2. Un constructeur avec un comparateur spécifique qui définit un ordre personnalisé pour le tri des éléments via un objet **comparator**
  3. Toute autre collection (par exemple, une List) pour initialiser cet ensemble avec des valeurs initiales.

1
```
Set<Integer> set = new TreeSet<>();
```
Ordre naturel

2
```
Set<Integer> set1 = new TreeSet<>(Comparator.reverseOrder());
```
Ordre naturel inversé

3
```
Set<Integer> set3 = new TreeSet<>(List.of(1,2,3));
```

COLLECTIONS : MAP

L'INTERFACE MAP

- **Map <K, V>** : **On utilise une Map lorsque l'on souhaite identifier une valeur à l'aide d'une clé.**
  - Lorsque vous utilisez le répertoire de contacts dans votre téléphone, vous cherchez le **nom** du contact (ex. "Ahmed") plutôt que de parcourir chaque **numéro** de téléphone un par un.

    (nom du contact, numéro de téléphone)
  - Lorsque vous utilisez un dictionnaire, vous cherchez le **mot** (ex. "arbre") pour identifier sa **définition.**

    (mot, définition)
- Une **Map** stocke donc des **paires clé-valeur => (K: key, V: value)**
  - **Les clés sont uniques**, mais les valeurs peuvent être dupliquées.
- Une **Map** est une composition d'un ensemble (**Set**) de clés et d'une collection (**Collection**) de valeurs.

MAP

- La méthode statique **`Map.ofEntries(…)`** permet d'initialiser une **map non modifiable** par des paires (clé, valeur).

```
Map<String,String> animaux = Map.ofEntries(
    Map.entry("Koala", "bamboo"),
    Map.entry("Lion", "viande"),
    Map.entry("Giraffe", "Feuilles")
);
```

- La méthode statique **`Map.copyOf(…)`** crée une copie non modifiable d'une Map existante.

```
Map<String,String> zoo = Map.copyOf(animaux);
```

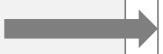
MAP

- L'interface **Map** fournit des méthodes spécifiques pour manipuler à la fois les clés et les valeurs.
  - `V put(K key, V value)` : **Ajoute ou remplace une paire clé/valeur**. Retourne la valeur précédente ou null.
  - `V putIfAbsent(K key, V value)`: **Ajoute la paire clé/valeur si la clé n'existe pas . Sinon, retourne la valeur existante.**
  - `V get(Object key)` : **Retourne la valeur associée à la clé, ou null si aucune valeur n'est associée.**
  - `boolean containsKey(Object key)` : **Vérifie si la clé est présente dans une map.**
  - `boolean containsValue(Object value)` : **Vérifie si la valeur est présente dans une map.**
  - `V remove(Object key)`: **supprime la paire clé/valeur et retourne la valeur associée à la clé. Retourne null sinon.**
  - `V replace(K key, V value)`: **Remplace la valeur associée à la clé donnée si la clé est présente. Retourne l'ancienne valeur ou null sinon.**
  - `Set<K> keySet()`: **Retourne l'ensemble de toutes les clés.**
  - `Collection<V> values()` : **Retourne une collection de toutes les valeurs.**
  - `Set<Map.Entry<K,V>> entrySet()`: **Retourne un ensemble de paires clé/valeur.**

MAP

```
Map<String,String> animaux = Map.ofEntries(
     Map.entry("koala", "bamboo"),
     Map.entry("lion", "viande"),
     Map.entry("giraffe", "feuilles")
);
```

```
public static void main(String[] args) {
…
     Map<String,String> zoo = Map.copyOf(animaux);

     System.out.println(zoo.containsKey("lion")); // true
     System.out.println(zoo.containsValue("lion")); // false
     System.out.println(zoo.size()); // 3
     System.out.println (zoo.get("koala")); // bamboo
     System.out.println (zoo.get("singe")); // null
     // Affichage avec forEach
     zoo.forEach((k,v) -> System.out.println(k +  ":" + v));
}
```



```
koala : bamboo
girafe : feuilles
lion : viande
```

IMPLÉMENTATIONS DU MAP

- **`HashMap`**, **`LinkedHashMap`** et **`TreeMap`** sont les trois classes qui implémentent l'interface **`Map`**.

Implémentation	Stockage des éléments	Caractéristiques	Quand l'utiliser ?
`HashMap<K, V>`	Table de hachage	Rapide mais ordre arbitraire, utilisant le hashCode() de la clé.	Stockage rapide de clé-valeur
`TreeMap<K, V>`	Arbre binaire	Clés triées, ordre des clés	Avoir un ordre sur les clés
`LinkedHashMap<K, V>`	HashMap + liste chaînée	Ordre d'insertion conservé	Parcours dans l'ordre d'insertion

IMPLÉMENTATIONS DU MAP

```
static void ajouterAnimaux(Map<String, String> zoo) {
    zoo.put("koala", "bamboo");
    zoo.put("lion", "viande");
    zoo.put("giraffe", "feuilles");
        for (String key : zoo.keySet())
        System.out.print(key + ', ');
}
```

zoo.keySet() : retourne l'ensemble des clés du map

```
public static void main(String[] args) {
    ajouterAnimaux(new HashMap<>()); // koala, giraffe, lion,
    ajouterAnimaux(new LinkedHashMap<>()); // koala, lion, giraffe,
    ajouterAnimaux(new TreeMap<>()); // giraffe, koala, lion,
}
```

Ordre aléatoire

Ordre d'insertion

Ordre des clés

IMPLÉMENTATIONS DU MAP

```
public static void main(String[] args) {

    Map<String,String> zoo = new HashMap<>();
    zoo.put("koala", "bamboo");
    zoo.put("lion", "viande");
    zoo.put("giraffe", "feuilles");
    System.out.println(zoo.replace("lion"," poulet ")); // viande
    System.out.println(zoo.putIfAbsent("koala","banane")); // null
    System.out.println(zoo.remove("lion")); // poulet
    zoo.forEach((k,v) -> System.out.println(k + ":" + v));
    zoo.clear(); // vider zoo
    System.out.println(zoo.isEmpty()); // true
}
```

```
koala : bamboo
girafe : feuilles
```

IMPLÉMENTATIONS DU MAP

```
public static void main(String[] args) {
Map<String, Double> notes = new HashMap<>();
    notes.put("E100", 12.5); notes.put("E200", 16.0);  notes.put("E300", 10.0);
    //parcourir les clés
 for (String matricule : notes.keySet())
      notes.put(matricule, notes.get(matricule) + 2);
    //parcourir les valeurs
 for(Double note: notes.values())
        System.out.println(note);
//parcourir les paires (Key, value)
for (Map.Entry<String, Double> e : notes.entrySet())
   { System.out.println(e.getKey()+ " : " + e.getValue()); }
}
```

TREEMAP : ORDRE ET TRI

- **Une `TreeMap` trie automatiquement les entrées en fonction de leurs clés.**
- Lorsque les clés sont des objets d'une classe que vous avez créée, ces **clés doivent être comparables** pour que la **`TreeMap`** puisse les ranger dans le bon ordre.
  - Si on souhaite un ordre naturel, Il est nécessaire que cette classe **implémente l'interface Comparable** et **redéfinisse la méthode compareTo qui spécifie l'orde naturel.**
  - Si on souhaite un ordre différent de celui défini par **compareTo** , on peut alors fournir un **Comparator** personnalisé à la **`TreeMap`** exactement comme on le fait avec le **`TreeSet`**.

TREEMAP : ORDRE ET TRI

```
class Colis implements Comparable<Colis> {
    int numero;
    double poids;

    public Colis(int numero, double poids) {
    this.numero = numero;
    this.poids = poids;
    }
    @Override
    public int compareTo(Colis autre) {
    // Ordre naturel par numéro (ordre croissant)
    return Integer.compare(this.numero, autre.numero);
    }
    public int getPoids() {
    return poids;
    }
    @Override
    public String toString() {
    return "Colis{" + "n°=" + numero + ", poids=" + poids + '}';
    }
}
```

TREEMAP : ORDRE ET TRI

```
public static void main(String[] args) {

TreeMap<Colis, String> expeditions = new TreeMap<>(); // tri selon l'ordre naturel

    // les colis seront triés par numéro
    expeditions.put(new Colis(102, 4.2), "Tanger");
    expeditions.put(new Colis(100, 2.5), "Casa");
    expeditions.put(new Colis(105, 3.8), "Rabat");

    expeditions.forEach((k,v)-> System.out.println (k + "->" + v));

}
```

```
Colis{ n°= 100, poids= 2.5 } -> Casa
Colis{ n°= 102, poids= 4.2 } -> Tanger
Colis{ n°= 105, poids= 3.8 } -> Rabat
```

TREEMAP : ORDRE ET TRI

```
//Comparator pour trier les colis par poids
class ComparateurParPoids implements Comparator<Colis> {
    @Override
    public int compare(Colis c1, Colis c2) {
        return Double.compare(c1.getPoids(), c2. getPoids());
    }
}
```

TREEMAP : ORDRE ET TRI

```
public static void main(String[] args) {

TreeMap<Colis, String> expeditions = new TreeMap<>(new ComparateurParPoids());

    // les colis seront triés par poids

    expeditions.put(new Colis(102, 4.2), "Tanger");
    expeditions.put(new Colis(100, 2.5), "Casa");
    expeditions.put(new Colis(105, 3.8), "Rabat");

    expeditions.forEach((k,v)-> System.out.println (k + "->" + v));

}
```

```
Colis{ n°= 100, poids= 2.5 } -> Casa
Colis{ n°= 105, poids= 3.8 } -> Rabat
Colis{ n°= 102, poids= 4.2 } -> Tanger
```

TRIER UNE COLLECTION

- Si l'on veut utiliser la méthode **Collections.sort()** pour trier une collection qui ne trie pas automatiquement ses éléments (comme une liste), les objets contenus dans cette collection doivent être comparables :
  - Soit via l'ordre naturel (**Comparable**),
  - Soit en précisant un ordre personnalisé via un objet **Comparator**.

```
public static void main(String[] args) {

    List<Colis> liste = new ArrayList<>();
    liste.add(new Colis(102, 3.5));
    liste.add(new Colis(101, 5.0));
    liste.add(new Colis(103, 2.0));
    System.out.println("Tri naturel (par numéro) :");
    Collections.sort(liste);
    System.out.println("Tri personnalisé par poids:");
    Collections.sort(liste, new ComparateurParPoids());
    }
```