Université Sultan Moulay Slimane
Ecole Supérieure de Technologie Béni Mellal
Département ITG
DUT- IDIA 1 - S2
M122 - Ingénierie Logicielle avec Java et UML
Année universitaire: 2024-2025

Support de cours

Ingénierie logicielle avec Java et UML

Partie I: Programmation Orientée Objet en Java

Chapitre IV : Les Collections

Pr. Siham BAKKOURI
Email: s.bakkouri@usms.ma

Plan

- **INTRODUCTION AUX COLLECTIONS EN JAVA**
- **LES ENUMERATIONS**
- **LES INTERFACES : COLLECTION ET MAP**
- **LES OBJETS DES COLLECTIONS**
  - **LES LISTES (LIST)**
  - **LES ENSEMBLES (SET)**
  - **LES DICTIONNAIRES (MAP)**
- **LES ITÉRATEURS**
- **EXERCICES**

Introduction

- les **collections** sont des structures de données permettant de stocker, manipuler et organiser des objets.
- Les Collections en Java regroupent différents **types de données** organisés à travers des **interfaces**, des **implémentations** et des **algorithmes** pour répondre à divers besoins en termes de stockage et de manipulation de données :
  - **Types de données** : Listes, ensembles, cartes, etc.
  - **Interfaces** : Définissent les spécifications pour les structures de données (`List`, `Set`, `Map`, etc.).
  - **Implémentations** : Classes concrètes qui mettent en œuvre ces interfaces (`ArrayList`, `HashSet`, `TreeMap`, etc.).
  - **Algorithmes** : Proposent des opérations prédéfinies comme le tri, la recherche, etc.

Ensemble, ces collections fournissent une flexibilité et une variété pour stocker, manipuler et gérer les données de manière efficace dans Java.

Les énumérations

- **Les énumérations**, en Java, sont des types de données spéciaux qui permettent de définir des collections de constantes nommées.
- Elles sont déclarées à l'aide du mot-clé `enum` et sont souvent utilisées pour représenter un ensemble **fixe de valeurs constantes** liées.
- Les **énumérations** offrent la possibilité de définir des ensembles de constantes étroitement liées à un type particulier.

➔ Elles permettent de rendre le code plus **lisible**, plus sûr et plus facile à maintenir en définissant un ensemble restreint de valeurs possibles pour une variable.

Les énumérations

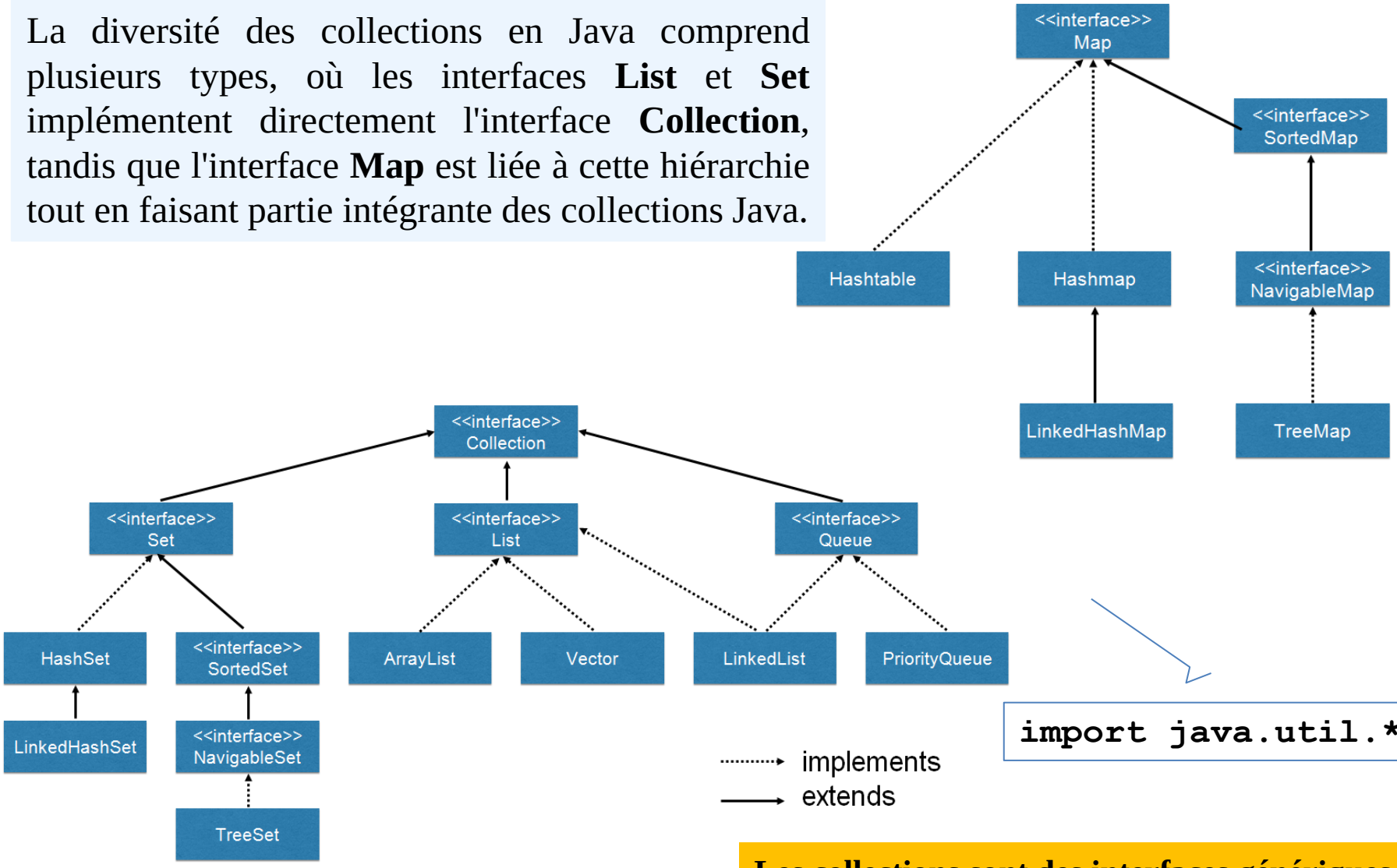
Exemple

```
enum JourSemaine {
    LUNDI,
    MARDI,
    MERCREDI,
    JEUDI,
    VENDREDI,
    SAMEDI,
    DIMANCHE
}

public class Main {
    public static void main(String[] args) {
        JourSemaine jour = JourSemaine.MERCREDI;
        System.out.println("Le jour de la semaine est : " + jour);
    }
}
```

Les interfaces: Collection et Map

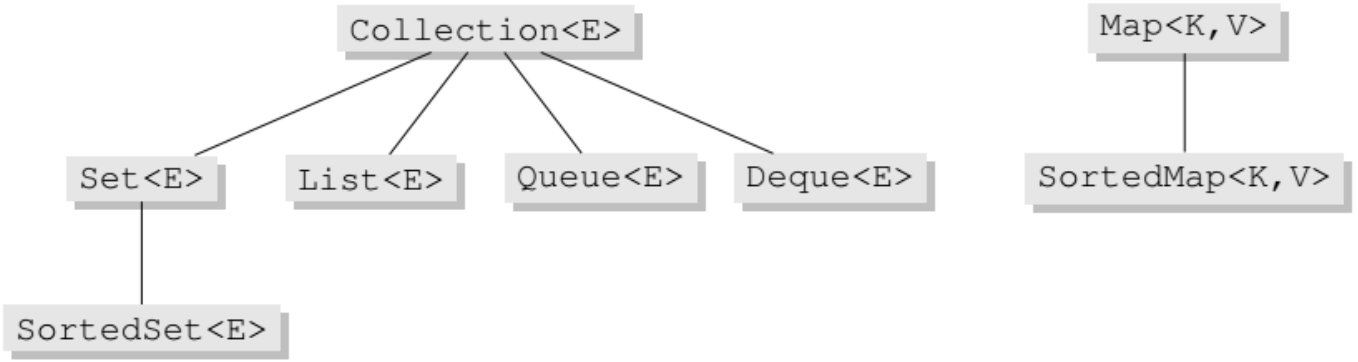
La diversité des collections en Java comprend plusieurs types, où les interfaces **List** et **Set** implémentent directement l'interface **Collection**, tandis que l'interface **Map** est liée à cette hiérarchie tout en faisant partie intégrante des collections Java.




```
import java.util.*;
```

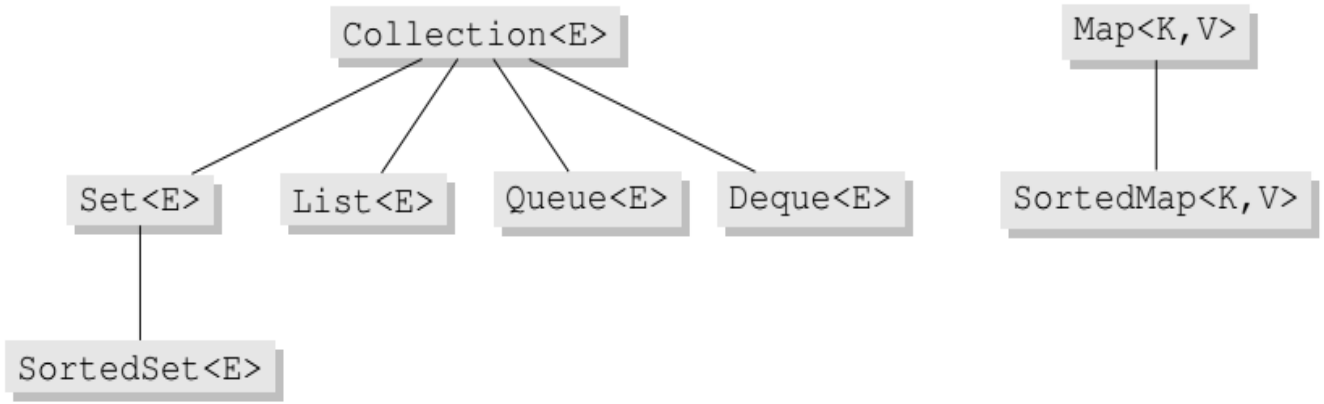
Les collections sont des interfaces génériques

Les interfaces: Collection et Map



- **Collection**<E> :propose des méthodes fondamentales pour parcourir, ajouter et supprimer des éléments, telles que **add**, **remove**, **size**, **toArray**, etc.
- **Set**<E> : stocke des éléments sans duplication.
- **SortedSet**<E> : représente une version triée des ensembles.
- **List**<E> : stocke des éléments dans un ordre spécifique et peut contenir des doublons.
- **Queue**<E> : représente une file d'attente où l'ordre d'ajout et de retrait des éléments est crucial, autorisant les doublons.
- **Deque** <E> : cette interface ressemble à une file d'attente, mais met l'accent sur les éléments à la fois en tête et en queue.

Les interfaces: Collection et Map



- **Map**<K,V> : associe des clés à des valeurs, représentant une relation binaire (surjective) où chaque élément est lié à une clé unique, bien que des doublons puissent exister pour les valeurs.
- **SortedMap**<K,V> : une version comportant des clés triées.

De plus

- **Iterator**<E> : une interface permettant de parcourir successivement les éléments via les méthodes ***next(), hasNext(), remove().***
- **ListIterator**<E> : un itérateur spécifique pour les List avec des opérations telles que ***set(E), previous, add(E).***

Les interfaces: Collection et Map

Exemple

```java
import java.util.*;
public class ExempleListeVehicules {
    public static void main(String[] args) {
        // Création d'une liste de véhicules (noms de véhicules)
        List<String> listeVehicules = new ArrayList<>();

        // Ajout de véhicules à la liste
        listeVehicules.add("Toyota Corolla");
        listeVehicules.add("Honda Civic");
        listeVehicules.add("Ford Focus");

        // Affichage des véhicules dans la liste
        System.out.println("Liste des véhicules :");
        for (String vehicule : listeVehicules) {
            System.out.println(vehicule);
        }

        // Suppression d'un véhicule de la liste (par exemple le deuxième véhicule)
        listeVehicules.remove(1);

        // Affichage de la liste après suppression
        System.out.println("\nListe des véhicules après suppression :");
        for (String vehicule : listeVehicules) {
            System.out.println(vehicule);
        }
    }
}
```

Les objets des collections

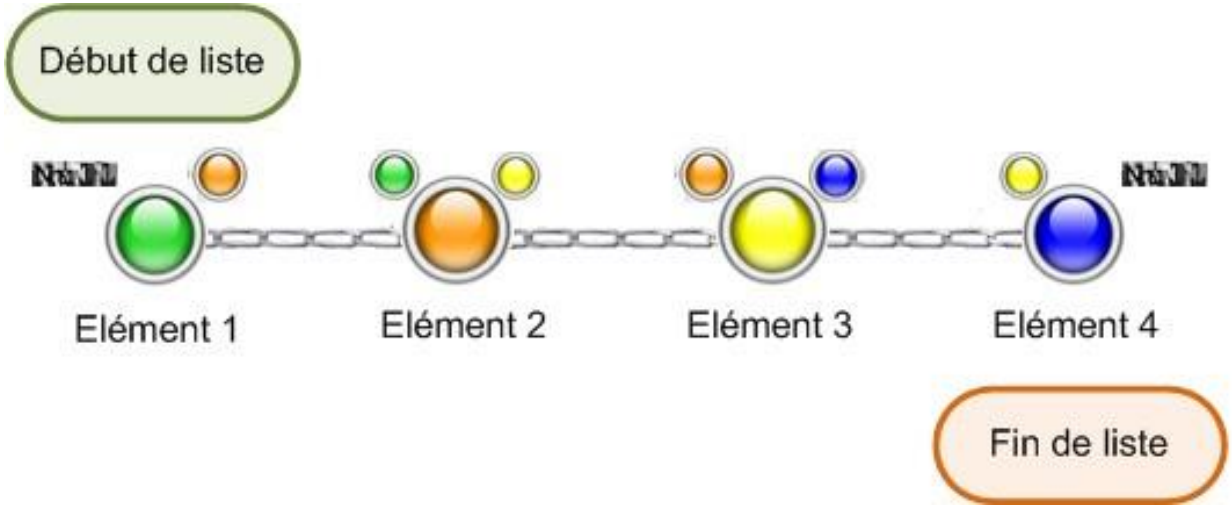
→ Les objets: List

- Les objets de la catégorie **List** sont des structures de données qui agissent comme des tableaux extensibles.
- Ces structures permettent d'ajouter un nombre illimité d'éléments sans se préoccuper de la taille initiale du tableau. Leur fonctionnement repose sur la possibilité de récupérer des éléments en utilisant leurs indices.
- Les **List** offrent une souplesse dans la gestion des données, permettant l'insertion et la récupération d'éléments selon les besoins du programme.
- De plus, les **List** contiennent des objets, ce qui les rend polyvalentes pour stocker différents types de données.

Pour explorer davantage cette catégorie, il est intéressant de se pencher sur deux objets spécifiques, **LinkedList** et **ArrayList**, qui peuvent s'avérer très utiles dans divers contextes.

→ Les objets: List
→ LinkedList

- **LinkedList** est une structure de données représentant une liste chaînée où chaque élément est connecté aux éléments adjacents par le biais de références.
- Chaque élément contient une référence à l'élément **précédent** et à l'élément **suivant**, sauf pour le premier élément dont la référence précédente vaut `Null`, et le dernier élément dont la référence suivante vaut également `Null`.



Les objets des collections

→ Les objets: List

→ LinkedList

Exemple

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

public class GInfo {

 public static void main(String[] args) {
   List listeC = new LinkedList();
   listeC.add(45.55f);
   listeC.add("ESTBM");
   listeC.add(123);

   for(int i = 0; i < listeC.size(); i++)
     System.out.println("Élément à l'index " + i + " = " + listeC.get(i));
     //Parcours avec un itérateur
     System.out.println("\n Parcours avec un itérateur ");
     System.out.println("--------------------------------");
     ListIterator li = listeC.listIterator();

     while(li.hasNext())
       System.out.println(li.next());
 }
}
```

Les objets des collections

→ **Les objets: List**
→ **LinkedList**

- Il y a une autre caractéristique importante à connaître concernant ce type d'objet : ils implémentent l'interface **Iterator**. Ainsi, cette interface peut être utilisée pour parcourir notre **LinkedList (**comme cela a été illustré dans l'exemple précédent**).**

  ➔ Un **itérateur** est un objet conçu spécifiquement pour parcourir une collection de données. C'est d'ailleurs sa principale fonction.

Remarque: vu que tous les éléments contiennent une référence à l'élément suivant, de telles listes risquent de devenir particulièrement lourdes en grandissant

→ Cependant, elles sont adaptées lorsqu'il faut beaucoup manipuler une collection en supprimant ou en ajoutant des objets en milieu de liste. Elles sont donc à utiliser avec précaution.

→ Les objets: List
→ ArrayList

- **ArrayList** est un objet très utile en programmation. Il s'agit d'une structure de données sans limite de taille, capable de stocker divers types de données, y compris la valeur **`null`**.

- Cet objet offre une grande souplesse, permettant l'ajout de tout type d'élément.

Exemple:

```
import java.util.ArrayList;
public class Test {

 public static void main(String[] args) {

   ArrayList al = new ArrayList();
   al.add(12);
   al.add("Une chaîne de caractères !");
   al.add(12.20f);
   al.add('d');

   for(int i = 0; i < al.size(); i++)
   {
     System.out.println("donnée à l'indice " + i + " = " + al.get(i));
   }
 }
}
```

→ Les objets: List
→ ArrayList

Les **ArrayList** présentent plusieurs avantages. Il existe une gamme étendue de méthodes disponibles avec cet objet :

- `add()` **:** permet d'ajouter un élément
- `get(int index)` **:** retourne l'élément à l'indice spécifié
- `remove(int index)` **:** supprime l'entrée à l'indice indiqué
- `isEmpty()` **:** renvoie "**true**" si l'objet est vide
- `removeAll()` **:** efface tout le contenu de l'objet
- `contains(Object element)` **:** retourne "**true**" si l'élément spécifié est présent dans l'ArrayList

Les objets des collections

→ Les objets: List
→ LinkedList vs. ArrayList

- Contrairement aux **LinkedList**, les **ArrayList** sont rapides en lecture, même avec un grand volume d'objets.
- Cependant, elles sont moins performantes lorsqu'il s'agit d'ajouter ou de supprimer des données au milieu de la liste.

➔ si vous effectuez principalement des lectures sans tenir compte de l'ordre des éléments, optez pour un **ArrayList** ; cependant, si vous insérez fréquemment des données au milieu de la liste, préférez une **LinkedList**.

Les objets des collections

→ Les objets: Set

- Un **Set** est une collection qui ne permet pas la présence de doublons.
  → Par exemple, elle n'autorise qu'une seule occurrence de **`null`**, car deux valeurs **`null`** seraient considérées comme un doublon.
- Les **Set** sont particulièrement adaptés pour manipuler des ensembles de données sans doublons. Cependant, leurs performances peuvent être affectées lors des opérations d'insertion.
- Parmi les implémentations de **Set**, on retrouve des objets tels que **HashSet**, **TreeSet**, **LinkedHashSet**, entre autres. Certains types de **Set** sont plus restrictifs que d'autres : certains n'acceptent pas la valeur **`null`**, certains types d'objets, etc.
- En générale, le **HashSet** est souvent préféré en raison de son accès plus rapide aux éléments. Cependant, si vous avez besoin d'une collection constamment triée, il est conseillé d'utiliser un **TreeSet**.

→ Les objets: Set
→ HashSet

L'objet **HashSet** est l'une des implémentations les plus utilisées de l'interface **Set**. Il est possible de parcourir cette collection à l'aide d'un objet **Iterator** ou d'extraire un tableau d'**Objects** à partir du **HashSet**, facilitant ainsi la manipulation et l'utilisation des données stockées.

Exemple:

```
import java.util.HashSet;
import java.util.Iterator;

public class Test {
 public static void main(String[] args) {
   HashSet hs = new HashSet();
   hs.add("toto");
   hs.add(12);
   hs.add('d');

   Iterator it = hs.iterator();
   while(it.hasNext())
     System.out.println(it.next());

   System.out.println("\nParcours avec un tableau d'objet");
   Object[] obj = hs.toArray();
   for(Object o : obj)
     System.out.println(o);
 }
}
```

Les objets des collections

→ **Les objets: Set**

→ **HashSet**

Voici une liste des méthodes disponibles dans cet objet :

- `add()` **:** ajoute un élément à la collection
- `contains(Object value)` **:** retourne "**true**" si l'objet contient la valeur spécifiée
- `isEmpty()` **:** retourne "**true**" si la collection est vide
- `iterator()` **:** renvoie un objet de type **Iterator** pour parcourir la collection
- `remove(Object o)` **:** supprime l'objet spécifié de la collection
- `toArray()` **:** retourne un tableau d'**Objects** contenant les éléments de la collection

Les objets des collections

→ Les objets: Map

Type de collection : Map

- Fonctionne sur la base d'une association ***clé - valeur***. Exemples : ***Hashtable***, ***HashMap***, ***TreeMap***, ***WeakHashMap***, etc.
- Clé utilisée pour identifier une entrée est unique ; plusieurs clés peuvent pointer vers une même valeur.

Inconvénient principal : Dépendance à la taille des données

- Performance affectée par la quantité de valeurs à stocker dans un objet Map.
- Explication logique : stocke une information supplémentaire pour chaque enregistrement par rapport à d'autres collections.
- Surcharge de mémoire supplémentaire peut entraîner des ralentissements, malgré la capacité mémoire élevée des ordinateurs actuels.

Gestion mémoire dans le développement Java :

- Cruciale pour les applications Java.

Notamment critique pour les appareils avec des ressources mémoire limitées.

→ Les objets: Map
→ Hashtable

❑ Objet Hashtable

- Également appelé "table de hachage".
- Parcours via les clés qu'il contient en utilisant la classe **`Enumeration`**.
- **Enumeration** : Contient **`Hashtable`** et facilite son parcours.

❑ Utilisation de l'énumération

- Exemple de code insérant les quatre saisons avec des clés non consécutives.
- L'énumération récupère uniquement les valeurs, démontrant ainsi son fonctionnement dans cet extrait de code.

Les objets des collections

→ Les objets: Map

→ Hashtable

Exemple:

```
import java.util.Enumeration;
import java.util.Hashtable;

public class Test {

 public static void main(String[] args) {

   Hashtable ht = new Hashtable();
   ht.put(1, "printemps");
   ht.put(10, "été");
   ht.put(12, "automne");
   ht.put(45, "hiver");

   Enumeration e = ht.elements();

   while(e.hasMoreElements())
     System.out.println(e.nextElement());

 }
}
```

→ Les objets: Map
→ Hashtable

Cet objet offre plusieurs méthodes utiles :

- **`isEmpty()`** retourne « vrai » si l'objet est vide.
- **`contains(Object value)`** retourne « vrai » si la valeur est présente, similaire à **`containsValue(Object value)`.**
- **`containsKey(Object key)`** retourne « vrai » si la clé passée en paramètre est présente dans la Hashtable.
- **`put(Object key, Object value)`** ajoute le couple clé - valeur dans l'objet.
- **`elements()`** fournit une énumération des éléments de l'objet.
- **`keys()`** renvoie la liste des clés sous forme d'énumération.

→ Il est important de noter également qu'un objet **`Hashtable`** n'accepte pas la valeur **`null`** et qu'il est **Thread-Safe**, ce qui signifie qu'il peut être utilisé dans plusieurs threads simultanément sans risque de conflit de données.

→ Les objets: Map
→ HashMap

❑ **Différences entre HashMap et Hashtable :**

- Acceptation de la valeur **`null`** : **`HashMap`** accepte la valeur **`null`**, contrairement à **`Hashtable`**.
- **Thread-Safety** : HashMap n'est pas Thread-Safe, tandis que Hashtable l'est.

❑ **Similitudes entre HashMap et Hashtable :**

- Ces deux types d'objets de la classe **Map** sont considérés comme équivalents dans une certaine mesure.
- Malgré quelques différences notables, ils partagent des similitudes fonctionnelles en tant qu'implémentations de la structure de données Map.

Les itérateurs

En Java, un **itérateur** est un objet qui permet de parcourir une collection (comme une List, un Set ou une Map) de manière séquentielle sans exposer la structure interne de la collection.

❑ Pourquoi utiliser un itérateur ?

- Accéder aux éléments d'une collection un par un.
- Éviter les erreurs liées à l'utilisation d'une boucle classique (for ou while).
- Supprimer des éléments en toute sécurité pendant l'itération.

❑ Les types d'itérateurs en Java

Java fournit plusieurs interfaces d'itérateurs :

- **`Iterator<E>`**:Permet de parcourir une collection et de supprimer des éléments.
- **`ListIterator<E>`** :Etend **`Iterator`**, permet une navigation dans les deux sens (**`previous`**() et **`next`**()).
- **`Spliterator<E>`** Utilisé pour le traitement parallèle des collections.

Les itérateurs

❑ Utilisation de `Iterator`

L'interface **`Iterator<E>`** est la plus courante. Elle fournit trois méthodes principales :

- **`hasNext()`:**Retourne true s'il y a un élément suivant.
- **`next()`**:Retourne l'élément suivant et avance l'itération.
- **`remove()`**:Supprime l'élément actuel de la collection.

Exemple

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

public class ExempleIterator {
    public static void main(String[] args) {

        ArrayList<String> noms = new ArrayList<>();
        noms.add("Amina");
        noms.add("Assia");
        noms.add("Ali");

        // Création de l'itérateur
        Iterator<String> it = noms.iterator();

        // Parcours de la liste avec l'itérateur
        while (it.hasNext()) {
            String nom = it.next();
            System.out.println("Nom : " + nom);
        }
    }
}
```

Exercices

Exercice 1

Écris un programme en Java qui :

1. Déclare une ***`ArrayList<String>`*** pour stocker des noms.
2. Ajoute 5 noms à la liste.
3. Affiche tous les noms de la liste.
4. Supprime un nom donné par l'utilisateur.
5. Affiche la liste après suppression.

→ Correction

Exercice 2

1. Crée une classe ***`Produit`*** avec deux attributs : ***`nom`*** et ***`prix`***.
2. Crée une classe ***`Stock`*** qui gère une liste de produits avec :
   - Une méthode pour ajouter un produit.
   - Une méthode pour afficher tous les produits.
   - Une méthode pour rechercher un produit par son nom.
   - Une méthode pour supprimer un produit par son nom.
3. Tester votre code en :
   - Ajoutant quelques produits.
   - Affichant la liste des produits.
   - Supprimant un produit donné.

→ Correction

ANNEXE

Exercice 1

```java
import java.util.ArrayList;
import java.util.Scanner;

public class GestionNoms {
    public static void main(String[] args) {

        ArrayList<String> noms = new ArrayList<>();

        noms.add("Ali");
        noms.add("Sofia");
        noms.add("Karim");
        noms.add("Leila");
        noms.add("Omar");

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

        Scanner scanner = new Scanner(System.in);
        System.out.print("Entrez un nom à supprimer : ");
        String nomASupprimer = scanner.nextLine();

        if (noms.remove(nomASupprimer)) {
            System.out.println(nomASupprimer + " a été supprimé.");
        } else {
            System.out.println("Nom non trouvé.");
        }

        System.out.println("Liste mise à jour : " + noms);

        scanner.close();
    }
}
```

Exercice 2 (1/3)

```
import java.util.ArrayList;
import java.util.Scanner;

class Produit {
    String nom;
    double prix;

    public Produit(String nom, double prix) {
        this.nom = nom;
        this.prix = prix;
    }

    @Override
    public String toString() {
        return nom + " - " + prix + " €";
    }
}
```

Exercice 2 (2/3)

```
class Stock {
    private ArrayList<Produit> produits = new ArrayList<>();

    public void ajouterProduit(String nom, double prix) {
        produits.add(new Produit(nom, prix));
    }
    public void afficherProduits() {
        if (produits.isEmpty()) {
            System.out.println("Le stock est vide.");
        } else {
            System.out.println("Produits en stock :");
            for (Produit p : produits) {
                System.out.println(p);
            }
        }
    }
    public void rechercherProduit(String nom) {
        for (Produit p : produits) {
            if (p.nom.equalsIgnoreCase(nom)) {
                System.out.println("Produit trouvé : " + p);
                return;
            }
        }
        System.out.println("Produit non trouvé.");
    }
    public void supprimerProduit(String nom) {
        for (Produit p : produits) {
            if (p.nom.equalsIgnoreCase(nom)) {
                produits.remove(p);
                System.out.println(nom + " a été supprimé.");
                return;
            }
        }
        System.out.println("Produit non trouvé.");
    }
}
```

Exercice 2 (3/3)

```
public class GestionStock {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        Stock stock = new Stock();


        stock.ajouterProduit("Pomme", 2.5);
        stock.ajouterProduit("Banane", 1.2);
        stock.ajouterProduit("Orange", 3.0);


        stock.afficherProduits();


        System.out.print("Entrez le nom du produit à rechercher : ");
        String produitRecherche = scanner.nextLine();
        stock.rechercherProduit(produitRecherche);

        System.out.print("Entrez le nom du produit à supprimer : ");
        String produitASupprimer = scanner.nextLine();
        stock.supprimerProduit(produitASupprimer);

        stock.afficherProduits();

        scanner.close();
    }
}
```