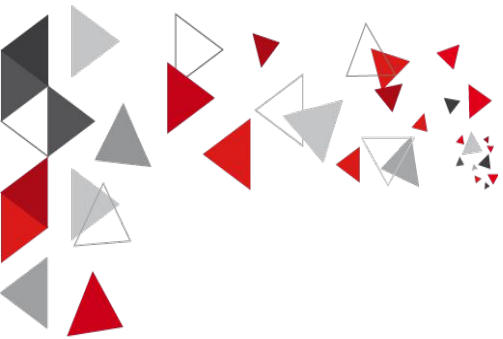
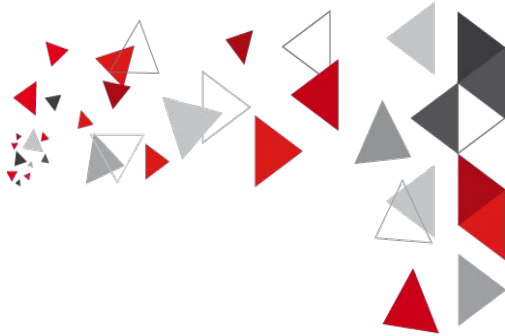


esprit
Se former autrement
HONORIS UNITED UNIVERSITIES

Conception Orienté Objet et Programmation Java

Chapitre 9 : Collection (List)

Objectifs du chapitre



- Découvrir les Interfaces fonctionnelles
- Découvrir la Collection List , son architecture et ses méthodes
- Exploiter l'utilité de la classe Collections
- Comprendre la différence entre Comparable et Comparator

Collection

Collection

En Java, une collection est **une structure de données** qui permet de stocker et de manipuler un groupe d'objets.

Les collections sont utilisés pour:

- stocker, rechercher et manipuler des données
- transmettre des données d’une méthode à une autre

Collection

Les collections en java sont un ensemble de **classes** et **d'interfaces** qui aident à manipuler les données efficacement.

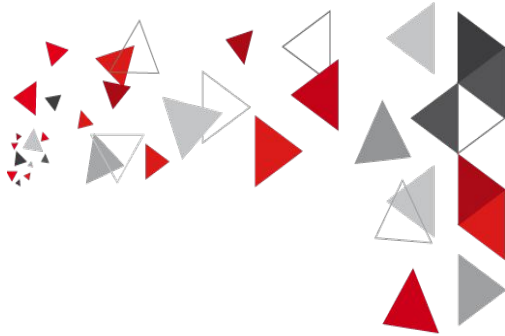
Il existe de nombreuses implémentations de collections :

- Certaines collections acceptent les doublons, d'autres pas.
- Certaines sont ordonnées, d'autres pas.
- Certaines collections émettent quelques restrictions, comme le type ou l'interdiction de la valeur null.

Collection VS Tableaux Statiques

	Collection	Tableau
Taille	Taille dynamique	Taille Fixe
Type d'element	Objet (pas de types primitifs)	Types primitifs et objet
Accès	Accès sequentiel	Accès par index
Manipulation	Des méthodes prédéfinies pour l'ajout , la suppression …	Créer des méthodes pour l'ajout, la suppression, …
Performance	Plus performants en terme de recherche ou le tri d'éléments.	Plus performants en terme d'accès et manipulation

Architecture

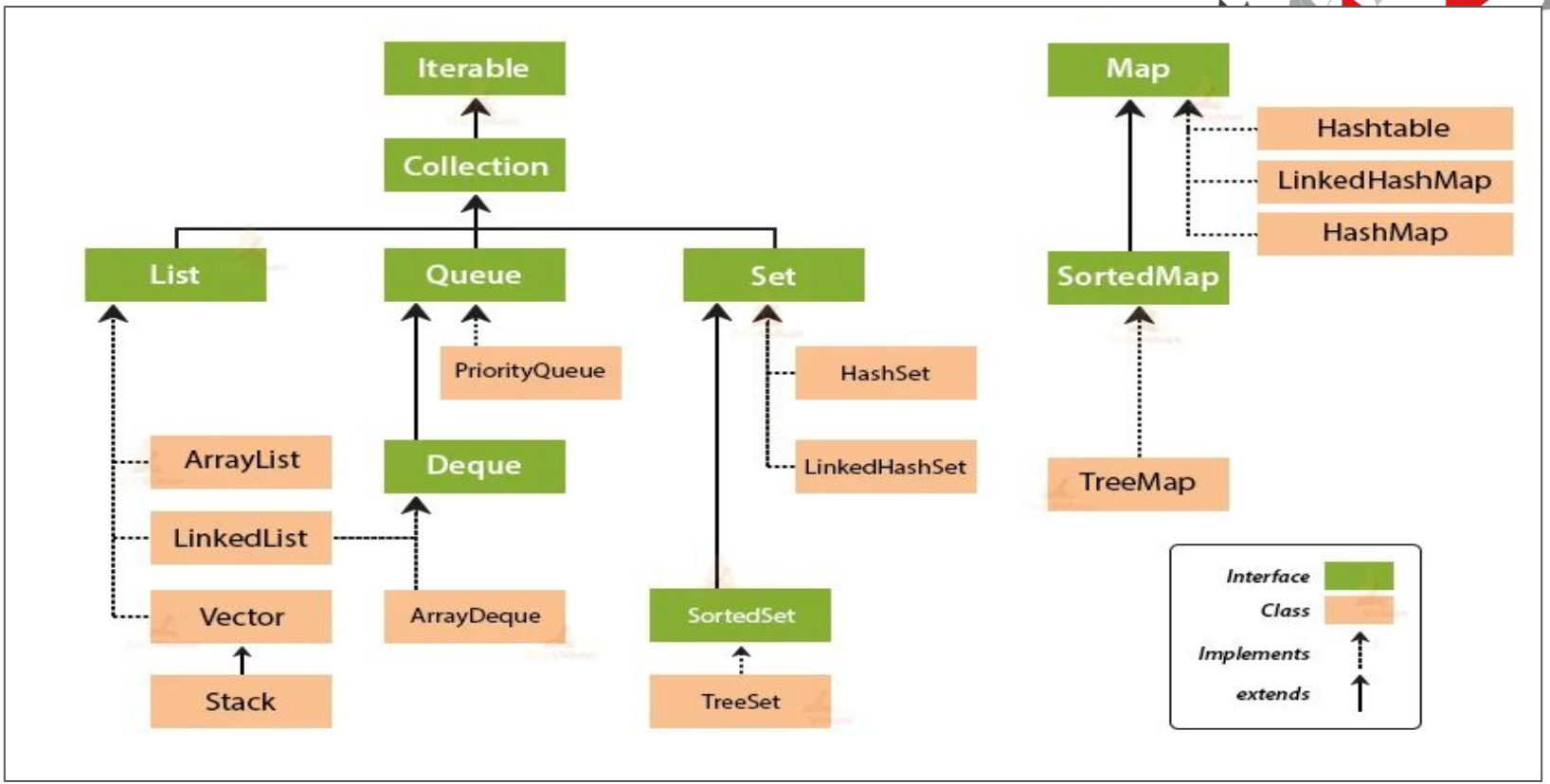


L’ architecture des collections est composé de 3 parties :

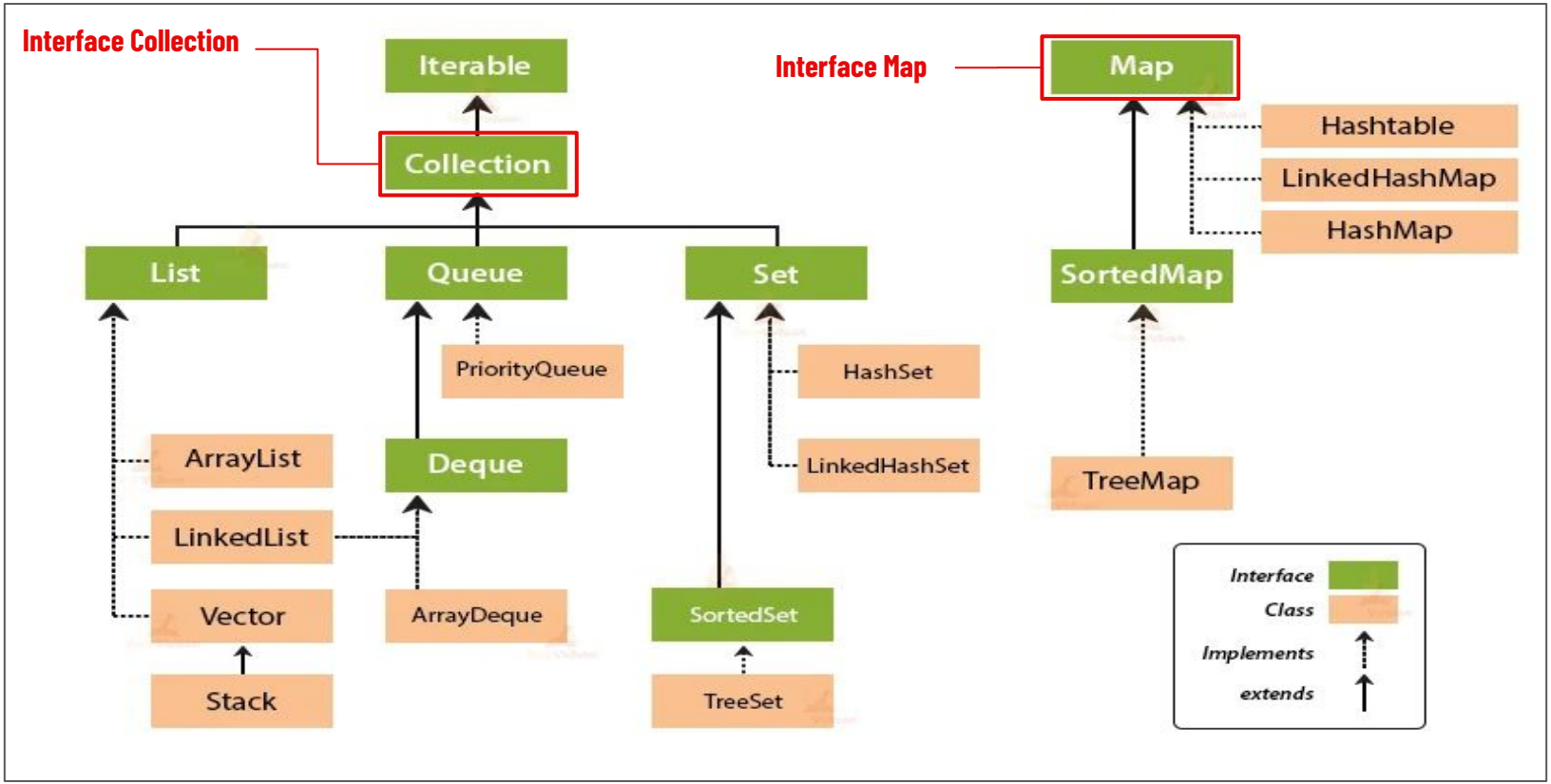
- Une hiérarchie d'interfaces permettant de représenter les collections sous forme de types abstraits.
- Des classes qui implémentent ces interfaces.
- Implémentation de méthodes liées aux collections (recherche, tri, etc.)

Ces classes et ces interfaces se trouvent dans le paquetage : java.util.

Architecture



Architecture



Architecture

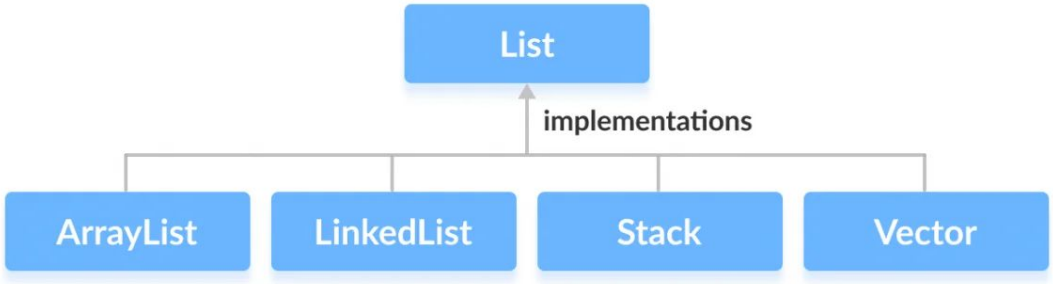
Méthode	Explication	Type de retour	Paramètres
add(E e)	Ajoute un élément à la fin de la collection.	boolean	e : l'élément à ajouter.
addAll(Collection<? extends E> c)	Ajoute tous les éléments de la collection spécifiée à la fin de cette collection.	boolean	c : la collection à ajouter.
clear()	Supprime tous les éléments de la collection.	void	N/A
contains(Object o)	Retourne true si la collection contient l'élément spécifié, sinon false.	boolean	o : l'élément à rechercher.
containsAll(Collection<?> c)	Retourne true si cette collection contient tous les éléments de la collection spécifiée, sinon false.	boolean	c : la collection à rechercher.
isEmpty()	Retourne true si la collection ne contient aucun élément, sinon false.	boolean	N/A
iterator()	Retourne un itérateur sur les éléments de la collection.	Iterator<E>	N/A

Architecture

Méthode	Explication	Type de retour	Paramètres
remove(Object o)	Supprime la première occurrence de l'élément spécifié de la collection, s'il est présent.	boolean	o : l'élément à supprimer.
removeAll(Collection<?> c)	Supprime tous les éléments de la collection qui sont également présents dans la collection spécifiée.	boolean	c : la collection à supprimer.
retainAll(Collection<?> c)	Supprime tous les éléments de la collection qui ne sont pas également présents dans la collection spécifiée.	boolean	c : la collection à conserver.
size()	Retourne le nombre d'éléments dans la collection.	int	N/A
toArray()	Retourne un tableau contenant tous les éléments de la collection, dans l'ordre où ils ont été insérés.	Object[ ]	N/A
toArray(T[ ] a)	Retourne un tableau contenant tous les éléments de la collection, dans l'ordre où ils ont été insérés, en les stockant dans le tableau spécifié si possible.	<T> T[ ]	a : le tableau dans lequel stocker les éléments.

List

List



En Java, **List** est une interface qui permet de stocker des éléments **ordonnés**.

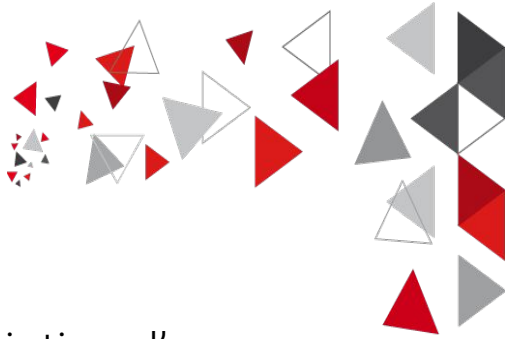
Chaque élément est accessible par sa position dans la liste, à l'aide d'un **indice** entier qui commence à 0 pour le premier élément.

Les classes qui implémente l'interface List sont des "tableaux" **extensibles** à volonté. On y trouve les classes **Vector, LinkedList et ArrayList**

List

	ArrayList	Vector
Implémentation	Implémente l'interface List, stockage dans un tableau dynamique	Implémente l'interface List, stockage dans un tableau dynamique synchronisé
Synchronisation	Non synchronisé	Synchronisé
Performances	Meilleures performances, car non synchronisé	Performances inférieures, en raison de la synchronisation
Capacité initiale	Peut spécifier la capacité initiale de la liste	Peut spécifier la capacité initiale de la liste
Augmentation de la taille	Augmentation de la taille de la liste par 50% de la capacité actuelle	Augmentation de la taille de la liste en double la taille de la capacité actuelle
Obsolescence	Non obsolète	Obsolète à partir de Java 1.2, mais toujours pris en charge

List : Déclaration



```
List monArrayList = new ArrayList(10);
```

Instanciation d'un Arraylist de reference List

```
List monVector = new Vector();
```

Instanciation d'un Vector de reference List

Remarque: L'attribution de la taille d'un tableau dynamique est facultatif !

List : Déclaration

Comme vous remarquez on utilise le **polymorphisme** par interface pour instancier une collection.

Cela nous permet de facilement passer à une implémentation de liste différente sans avoir à modifier le reste de votre code.

Par exemple, vous pouvez passer de **ArrayList** à **Vector** en modifiant simplement la partie instanciation.

Si vous utilisez **ArrayList** partout dans votre code, vous devriez modifier chaque instance si vous vouliez changer l'implémentation.

List : ArrayList

- Un **ArrayList** est un “tableau” qui se redimensionne automatiquement. Il accepte **tout type d'objets, null y compris.**

- Chaque instance d'ArrayList a une capacité, qui définit le nombre d'éléments qu'on peut y stocker.

- Au fur et à mesure qu'on ajoute des éléments et qu'on "dépasse" la capacité, la taille augmente en conséquence.

List : ArrayList

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

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

        // Création d'une liste générique qui accepte tous type de données
         List maListe = new ArrayList<>();

         maListe.add("Bonjour");
         maListe.add(8);
         maListe.add(true);
         maListe.add(null);

        // Une liste peut contenir des éléments redondants
         maListe.add("Bonjour");

         System.out.println(maListe);

    }
}
```

List : ArrayList

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

public class ListeExemple {
    public static void main(String[] args) {
        // Création d'une liste de chaînes de caractères de taille 2
        List<String> maListe = new ArrayList<>(2);

        // Ajout de 3 éléments à la liste , la liste se redimensionne
        maListe.add("Bonjour");
        maListe.add("à");
        maListe.add("tous");

        // Accès à un élément de la liste par son index
        String deuxiemeElement = maListe.get(1);
        System.out.println("Deuxième élément : " + deuxiemeElement);

        // Modification d'un élément de la liste
        maListe.set(2, "monde");
        System.out.println("Liste modifiée : " + maListe);
    }
}
```

List : ArrayList

```
public class ListeExemple {
    public static void main(String[] args) {
        // Création d'une liste de chaînes de caractères de taille 2
        List<String> maListe = new ArrayList<>(2);

        // Ajout de 3 éléments à la liste , la liste se redimensionne
        maListe.add("Bonjour");
        maListe.add("à");
        maListe.add("tous");
        // Suppression du premier élément de la liste
        maListe.remove(0);
        System.out.println("Liste modifiée : " + maListe);

        // Recherche d'un élément dans la liste
        System.out.println(maListe.contains("Bonjour"));
    }
}
```

Pour afficher les éléments d'une liste avec l'instruction "System.out.println()" , la redéfinition de la méthode toString() est nécessaire.

List : ArrayList

```
public class ListeExemple {

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

        maListe.add("Bonjour");
        maListe.add("à");
        maListe.add("tous");

        // Suppression du premier élément de la liste
        maListe.remove(0);

        // Recherche d'un élément dans la liste
        System.out.println(maListe.contains("Bonjour"));
    }
}
```

Pour rechercher ou supprimer un élément , la redéfinition de la méthode equals(Object obj) est nécessaire pour comparer deux objets selon leurs attributs.

Collection vs Collections

- **Collection** est une interface qui définit le comportement générique des collections de données en Java.

- **Collections** est une classe utilitaire qui possède des méthodes statiques pour manipuler des collections. Cette classe contient des méthodes qui permettent de **trier, rechercher, inverser, mélanger, etc.** les éléments d'une collection.

La classe Collections

Méthode	Description
copy(List<? super T> dest, List<? extends T> src)	Copie tous les éléments de la liste source vers la liste de destination
fill(List<? super T> list, T obj)	Remplit tous les éléments de la liste avec l'objet spécifié
max(Collection<? extends T> coll)	Retourne l'élément maximum de la collection donnée, selon l'ordre naturel des éléments
min(Collection<? extends T> coll)	Retourne l'élément minimum de la collection donnée, selon l'ordre naturel des éléments

La classe Collections

Méthode	Description
reverse(List<?> list)	Inverse l'ordre des éléments de la liste
shuffle(List<?> list)	Mélange les éléments de la liste dans un ordre aléatoire
sort(List<T> list)	Trie les éléments de la liste dans l'ordre naturel des éléments
swap(List<?> list, int i, int j)	Échange les éléments à la position i et j dans la liste

La méthode **Collections.sort(List l)** est une méthode statique fournie par la classe Collections, qui permet de trier les éléments d'une collection donnée dans l'ordre naturel des éléments.

Si la méthode **sort()** est utilisée, il faut obligatoirement que les éléments inclus dans la liste implémentent tous l'interface **Comparable** sinon une exception de type ***ClassCastException*** est levée.

La classe Collections (Tri)

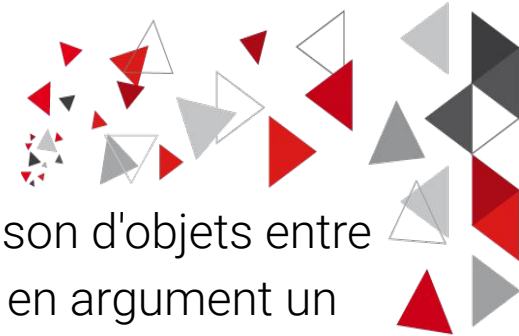
```java
public class ListeExemple {
    public static void main(String[] args) {
        List<String> fruits = new ArrayList<>();

        fruits.add("pomme");
        fruits.add("banane");
        fruits.add("orange");

       // Affichage de la liste non triée
       System.out.println("Liste non triée : " + maListe);

       // Tri de la liste en utilisant Collections.sort()
       Collections.sort(fruits);

       // Affichage de la liste triée
       System.out.println("Liste triée : " + fruits);
    }
}
```

Nb: La classe String implémente l'interface Comparable

Output :

```
Liste non triée : [pomme, banane, orange]
Liste triée : [banane, orange, pomme]
```

L’interface Comparable<T>



L’interface **Comparable** est utilisée pour permettre la comparaison d'objets entre eux. Cette interface définit une méthode **compareTo** qui prend en argument un **objet** et retourne un **entier**.

```
public interface Comparable<T> {
    public int compareTo(T o);
}
```

La méthode **compareTo** compare l'objet courant avec l'objet donné en argument et renvoie :

- un nombre négatif si l'objet courant est inférieur à l'objet donné en argument.
- zéro si l'objet courant est égal à l'objet donné en argument.
- un nombre positif si l'objet courant est supérieur à l'objet donné en argument.

- a.compareTo (b) == 0 si a.equals To(b)
- a.compareTo (b) < 0 si a plus « petit »que b
- a.compareTo (b) > 0 si a plus « grand » que b

Comparable : exemple(1/2)

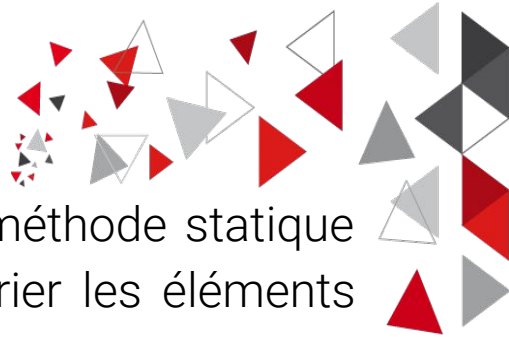
```
public class Personne implements Comparable<Personne> {

    private String nom;
    private int age;


    public Personne(String nom, int age) {
        this.nom = nom;
        this.age = age;
    }


    public int compareTo(Personne autre) {
        return this.age - autre.age;
    }

     @Override
     public String toString() {
          return "Personne{ nom= " + nom + ", age=" + age + " }";
     }
}
```

Comparable : exemple(2/2)

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

     List<Personne> personnes = new ArrayList<>();
     personnes.add(new Personne("Mohamed", 24));
     personnes.add(new Personne("Ali", 29));
     personnes.add(new Personne("Marwa", 21));

     System.out.println("Liste non triée: " + personnes);

     Collections.sort(personnes);

     System.out.println("Liste triée: " + personnes);

     }
}
```

Liste non triée: [Personne{ nom= Mohamed, age=24 }, Personne{ nom= Ali, age=29 }, Personne{ nom= Marwa, age=21 }]

Liste triée: [Personne{ nom= Marwa, age=21 }, Personne{ nom= Mohamed, age=24 }, Personne{ nom= Ali, age=29 }]



La méthode **Collections.sort(List l, Comparator c)** est une méthode statique fournie par la classe Collections qui offre la possibilité de trier les éléments d'une collection suivant des critères personnalisés spécifiés en tant que paramètres.

Cette fonctionnalité permet d'effectuer un tri sur une liste en fonction de plusieurs critères. Par exemple, vous pouvez trier une liste de personnes en utilisant d'abord l'âge comme critère principal, et en cas d'égalité d'âge, vous pouvez ensuite comparer les noms pour obtenir un tri plus précis et complexe.

L'interface Comparator

```
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

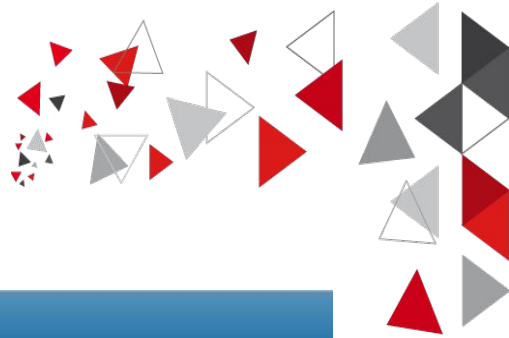
int compare(T o1, T o2) qui doit renvoyer

- un entier positif si **o1 est « plus grand » que o2**
- 0 si **o1 a la même valeur (au sens de equals)** que **o2**
- un entier négatif si **o1 est « plus petit » que o2**

Comparator : exemple

```
class PersonneNomComparator implements Comparator<Personne> {
    @Override
    public int compare(Personne p1, Personne p2) {
        return p1.getNom().compareTo(p2.getNom());
    }
}

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

     List<Personne> personnes = new ArrayList<>();
     personnes.add(new Personne("Mohamed", 24));
     personnes.add(new Personne("Ali", 29));
     personnes.add(new Personne("Marwa", 21));

     Collections.sort(personnes, new PersonneNomComparator());

     System.out.println("Liste triée selon le nom: " + personnes);

     }
}
```

Comparable vs Comparator



	Comparable	Comparator
Séquence de tri	Comparable fournit une séquence de tri unique. En d'autres termes, nous pouvons trier une collection sur la base d'un seul élément, tel que l'identifiant, le nom et l'adresse.	Comparator fournit plusieurs séquences de tri. En d'autres termes, nous pouvons trier la collection en fonction de plusieurs éléments tels que l'identifiant, le nom, l'adresse, etc.
La méthode	Comparable fournit la méthode **compareTo()** pour trier les éléments.	Comparator fournit la méthode **compare()** pour trier les éléments.
Package	Comparable existe dans le package java.lang.	Comparator existe dans le package java.util.
Trier	Nous pouvons trier les éléments d'une liste de type Comparable avec la méthode `Collections.sort(List)`.	Nous pouvons trier les éléments d'une liste de type Comparator avec la méthode `Collections.sort(List, Comparator)`

Interface fonctionnelle

Les interfaces fonctionnelles

Une **interface fonctionnelle** est une interface Java qui ne comporte qu'**une seule méthode abstraite**, permettant de représenter un comportement ou une fonction spécifique.

Elle est couramment utilisée pour définir des **expressions lambda** (prochain chapitre), facilitant ainsi la programmation en Java.

Une interface fonctionnelle pourra être annotée avec l'annotation

@FunctionalInterface

L'annotation **@FunctionalInterface** n'est pas obligatoire, elle a un rôle important permettant au compilateur de forcer l'interface afin qu'elle ne contienne qu'une seule méthode abstraite

```
@FunctionalInterface   //optionnel
interface Math1{

    int add(int x,int y);

    default int divide(int x,int y){
        return x / y;
    }
}
```

Si nous essayons de définir plus d'une seule méthode abstraite dans une interface annotée avec **@FunctionalInterface**, le compilateur affichera une erreur.

```
@FunctionalInterface   //erreur de compilation
interface Math1{

    int add(int x,int y);
    int multiply(int x,int y);

    default int divide(int x,int y){
        return x / y;
    }
}
```

Les interfaces fonctionnelles

Interface fonctionnelle	Description	Argument(s)	Type de retour	Méthode abstraite
Comparator<T>	compare entre deux objets de type T	T, T	int	compare(T t,1 T t2)
Supplier<T>	Fournit une valeur de type T.	Aucun	T	get()
Consumer<T>	Consomme une valeur de type T sans retour.	T	void	accept(T t)
BiConsumer<T, U>	Consomme deux valeurs de type T et U sans retour.	T, U	void	accept(T t, U u)
Function<T, R>	Transforme une valeur de type T en une valeur de type R.	T	R	apply(T t)

Les interfaces fonctionnelles

Interface fonctionnelle	Description	Argument(s)	Type de retour	Méthode abstraite
BiFunction<T, U, R>	Transforme deux valeurs de type T et U en une valeur de type R.	T, U	R	apply(T t, U u)
Predicate<T>	Vérifie si une valeur de type T satisfait une condition.	T	boolean	test(T t)
BiPredicate<T, U>	Vérifie si deux valeurs de type T et U satisfont une condition.	T, U	boolean	test(T t, U u)
UnaryOperator<T>	Transforme une valeur de type T en une valeur de type T.	T	T	apply(T t)
BinaryOperator<T>	Combine deux valeurs de type T en une seule valeur de type T.	T, T	T	apply(T t1, T t2)

Les interfaces fonctionnelles

Une interface fonctionnelle peut être instanciée à condition que vous fournissiez une implémentation de sa méthode abstraite.

Elles offrent une flexibilité accrue pour personnaliser les comportements en fournissant des implémentations de méthodes, **éliminant** ainsi la nécessité de créer de **plusieurs classes** qui implémentent l'interface.

Les interfaces fonctionnelles : exemple Comparator (1/3)

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

        List<Personne> personnes = new ArrayList<>();
        personnes.add(new Personne("Mohamed", 24));
        personnes.add(new Personne("Ali", 29));
        personnes.add(new Personne("Marwa", 24));

        Comparator<Personne> nomComparator = new Comparator<Personne>() {
            @Override
            public int compare(Personne o1, Personne o2) {
                return o1.getNom().compareTo(o2.getNom());
            }
        };

        Collections.sort(personnes, nomComparator);

        System.out.println("Liste triée selon le nom: " + personnes);
```

Instanciation à travers un objet anonyme

Les interfaces fonctionnelles : exemple Comparator (2/3)

```
        Comparator<Personne> ageComparator = new Comparator<Personne>() {
            @Override
            public int compare(Personne o1, Personne o2) {
                return o1.getAge() - o2.getAge();
            }
        };


        // Tri selon l'age, en cas d'égalité on compare selon le nom
        Collections.sort(personnes, ageComparator.thenComparing(nomComparator));

        System.out.println("Liste triée: " + personnes);

    }
}
```

Les interfaces fonctionnelles : exemple Comparator (3/3)

L'instanciation de l'interface via un **objet anonyme** signifie qu' on peut créer plusieurs implémentations de la méthode **compare** sans avoir à créer une classe dédiée pour chaque scénario spécifique. Cela simplifie considérablement le code en évitant la surcharge de classes.

Nous pouvons également **combiner** des comparateurs en utilisant la méthode **thenComparing** , qui est une méthode par défaut de l'interface **Comparator**. Cette approche nous permet de définir plusieurs critères de tri pour obtenir un ordre de tri plus complexe.

Merci pour votre attention