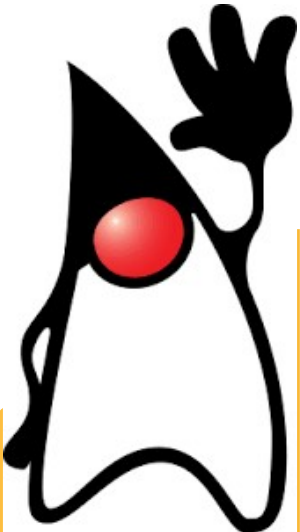




RÉPUBLIQUE TUNISIENNE

**MESRS**

Ministère de l'Enseignement Supérieur  
et de la Recherche Scientifique



# Cours Programmation Orientée Objets Avancée

Préparé par :

Hedi HMANI, Manel BEN SALAH, Sameh HBAIEB TURKI

Année universitaire  
2020-2021

# Chapitre 2: Généricité ET Collection





# Plan du chapitre

- **LA GÉNÉRICITÉ**

- ☐ Pourquoi la généricité
- ☐ Définition d'une classe générique simple
- ☐ Définition d'une méthode générique
- ☐ Règles d'héritage pour les types génériques
- ☐ Le type Jokers

- **LES COLLECTIONS**

- ☐ Avantages des collections Vs des Tableaux
- ☐ Listes ordonnées d'objets : ArrayList et LinkedList
- ☐ Ensembles d'objets uniques : HashSet et TreeSet
- ☐ Dictionnaires d'objets : HashMap et TreeMap



# **Partie 1: Généricité**



# Problématique

Une classe est composée d'un ensemble d'attributs + un ensemble de méthodes

Les attributs d'une classe ont **forcément** un nom et un type (*Java est fortement typé*)

Le type d'attribut ne change pas pour toutes les instances de cette classes,

Les méthodes permettent généralement d'effectuer des opérations sur les attributs tout en respectant leurs spécificités (type, taille...)

Et si on a besoin d'une classe dont les méthodes effectuent les mêmes opérations quel que soit le type d'attributs

- somme** pour **entiers** ou **réels**,
- concaténation** pour **chaînes de caractères**,
- ou **logique** pour **booléens**...

...

**Impossible sans définir plusieurs classes (une pour chaque type) et une interface ou en imposant le type Object et en utilisant plusieurs cast...**



# Problématique

**Exemple 1.** Nous allons créer la classe Paire qui permet de travailler avec n'importe quel type de donnée.

## Solution avec Object

```
public class Paire {  
    Object premier;  
    Object second;  
  
    public Paire(Object a, Object b) {  
        premier = a;  
        second = b;  
    }  
  
    public Object getPremier() {  
        return premier;  
    }  
  
    public Object getSecond() {  
        return second;  
    }  
}
```



## Echec de la solution

```
Paire p = new Paire ("abc", "xyz");  
String x = (String)p.getPremier();  
Double y = (Double)p.getSecond();  
  
// le casting est obligatoire  
// il faut attendre l'exécution pour avoir  
// une levée d'exception (ClassCastException)
```

Si nous voulons utiliser les données de l'objet *Paire*, il va falloir faire un **casting**. Certaines erreurs ne sont détectées qu'à l'**exécution** du programme (Casting du Type "Double" sur chaîne de caractère).



# Solution

Depuis Java 5, une solution plus élégante avec la **généricité**:

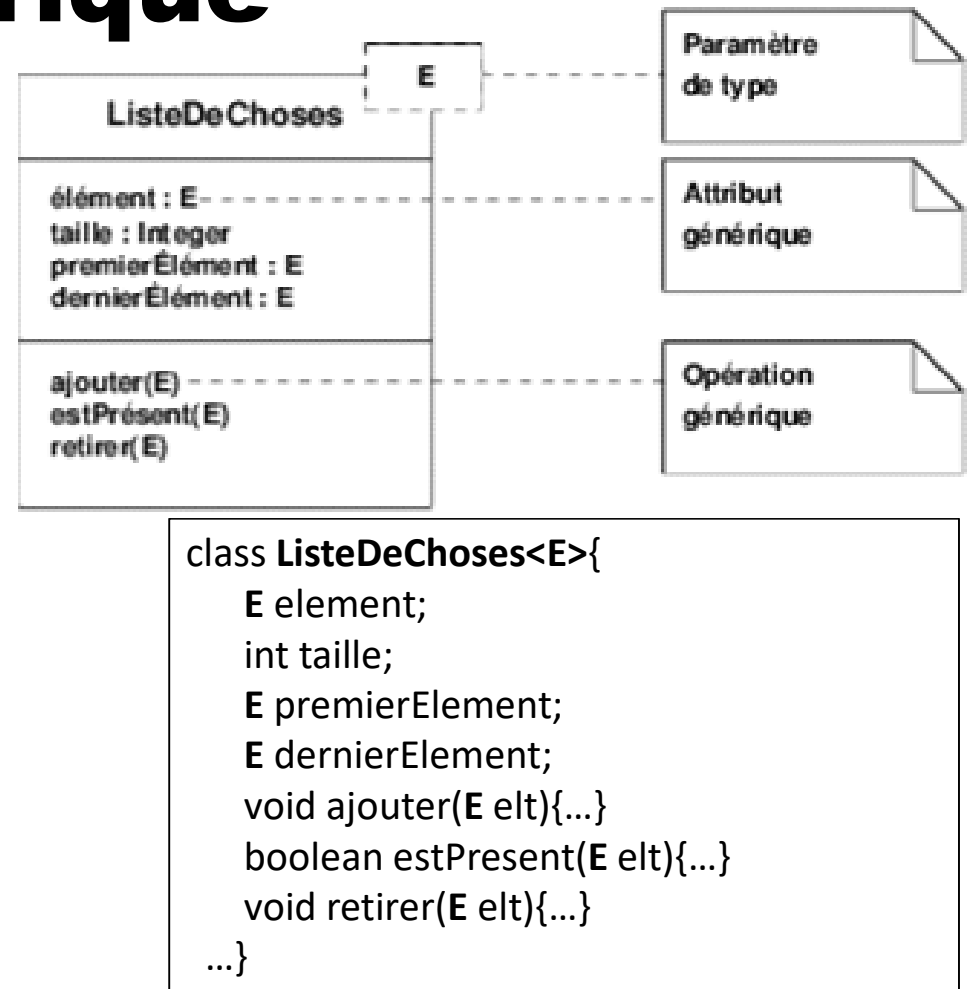
- Les classes paramétrées / génériques permettent de déclarer des classes qui contiennent des comportements génériques.
- On peut spécifier les attributs, une ou des opérations d'une classe avec des paramètres génériques
- Le type générique ne sera pas précisé à la création de la classe
- A l'instanciation de la classe, on précise le type à utiliser par cette classe

**On peut donc choisir pour chaque instance le type que l'on souhaite utiliser.**



# Définition – Classe paramétrée/générique

- ❑ Classe paramétrée / générique = paramétrée par des types
  - Attributs génériques = typés avec le type en paramètre
  - Opérations génériques = arguments et / ou type de retour génériques
- ❑ La généricité est présente dans de nombreux langages de programmation avant introduction en Java: Eiffel, Ada, C++, ...
- ❑ Présente dans le langage de modélisation UML





# Solution

**Exemple 1.** Nous allons reprendre notre classe Paire en la rendant générique :

Solution avec  
classe générique

```
public class Paire<T> {  
    T premier;  
    T second;  
    public Paire(T a, T b) {  
        premier = a;  
        second = b;  
    }  
    public T getPremier() {  
        return premier;  
    }  
    public T getSecond() {  
        return second;  
    }  
}
```



Solution avec  
succès

Le T n'est défini qu'à l'instanciation de la classe Paire

```
35 Paire<String> p = new Paire<String>("abc", "xyz");  
36 String x = p.getPremier(); // pas de cast  
37 Double y = p.getSecond(); // erreur de compilation (Type mismatch)  
..
```



Avec la Généricité, on a limité l'utilisation du casting (pas intuitif)

Les erreurs ont été détectées dès la compilation. Certaines erreurs ne sont détectées qu'à l'exécution du programme (Casting du Type "Double" sur chaîne de caractère) On peut simplement vérifier par un **instanceof** ou récupérer l'exception **ClassCastException**

# Pourquoi la généricité

- ❗ Transtypage (cast en anglais) à chaque fois que nous aurons besoins de récupérer un objet : `String x=((String)p.getPremier())`
- ❗ Il n'existe aucune procédure de vérification des erreurs.

## La Généricité permet d'écrire un code :

- ✔ Plus pratique
    - ✔ Plus simple
      - ✔ Plus sûr
        - ✔ Plus facile
          - ✔ Plus puissante
- ...

# Définition d'une classe générique simple

Une classe générique est une classe comprenant une ou plusieurs variables de TYPE : <T>

```
class Tableau<T>{  
    T[] tableau;  
    void affecterTab(T[] tabEntier){...}  
    T[] getTab(){...}  
    ...}
```

Tableau **<String>** obj1

Tableau **<Lion>** obj2

- obj1 est un objet de type Tableau<String>.
- Le paramètre de la classe Tableau est de type String.
- Tout ce qui fait référence à T est remplacé par String.

```
class Tableau{  
    String[] tableau;  
    void affecterTab(String[] tabEntier){...}  
    String[] getTab(){...}  
    ...}
```

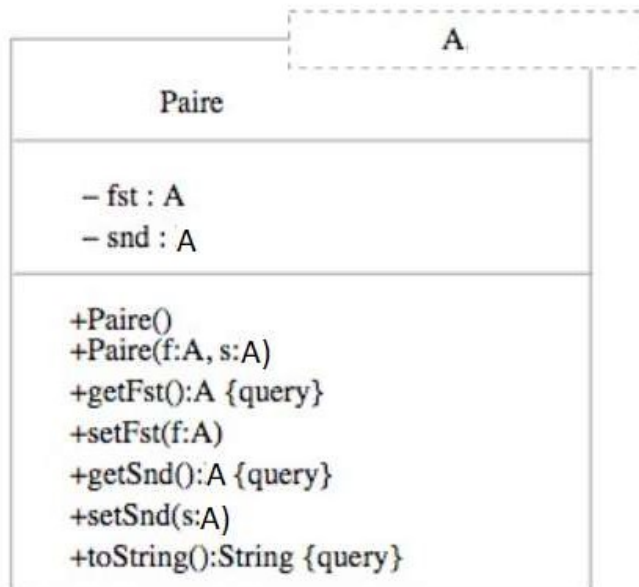
- obj2 est un objet de type Tableau<Lion>.
- Le paramètre de la classe Tableau est de type Lion.
- Tout ce qui fait référence à T est remplacé par Lion.

```
class Tableau{  
    Lion[] tableau;  
    void affecterTab(Lion[] tabEntier){...}  
    Lion[] getTab(){...}  
    ...}
```

# Exemple avec classe générique simple

**Exemple 1.** On veut écrire une classe paramétré Paire.

## Représentation en UML



Paire< A-> String > *Classes*

## Solution java

```
class Paire<A> {
    private A fst; private A snd;
    public Paire() {fst =null; snd =null;}
    public Paire(A fst, A snd) { this. fst = fst; this.snd = snd; }
    public A getFst() {return this.fst; }
    public void setFst(A fst) { this. fst = fst; }
    public A getSnd() {return this.snd; }
    public void setSnd(A snd) {this.snd = snd; }
}
```

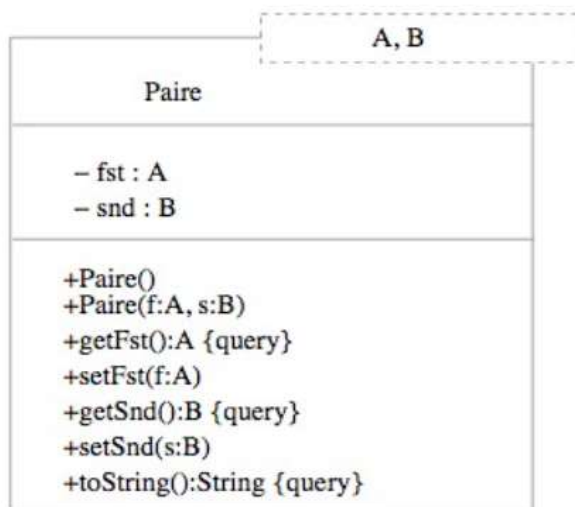
```
class TableauAlg {
    public static Paire<String> minmax(String[] chaînes) {
        if (chaînes==null || chaînes.length==0) return null;
        String min = chaînes[0]; String max = chaînes[0];
        for (String chaîne : chaînes) {
            if (min.compareTo(chaîne) > 0) min = chaîne;
            if (max.compareTo(chaîne) < 0) max = chaîne;
        } return new Paire<String>(min, max); }
}
```

```
public class Test {
    public static void main (String[] args) {
        String[] phrase = {"Marie", "possède", "une", "petite", "lampe"};
        Paire<String> extrêmes = TableauAlg.minmax(phrase);
        System.out.println("min = "+extrêmes.getFst());
        System.out.println("max = "+extrêmes.getSnd()); }
}
```

# Exemple avec classe à deux types génériques

**Exemple 2.** On veut écrire une classe paramétré Paire à deux types génériques.

## Représentation en UML



*Modèles de classes*

Paire< A-> Integer, B -> String >

*Classes*

## Solution java

```
class Paire<A,B> {
    private A fst; private B snd;
    public Paire() {fst =null; snd =null;}
    public Paire(A fst, B snd) { this. fst = fst; this.snd = snd; }
    public A getFst() {return this.fst; }
    public void setFst(A fst) { this. fst = fst; }
    public B getSnd() {return this.snd; }
    public void setSnd(B snd) {this.snd = snd; }
}
```

L'utilisation demande du *typecast*.

```
Paire p1= new Paire (27,"jour");
if(p1.getSnd() instanceof String)
    String p1Snd = (String) p1.getSnd();
```

# Méthode générique

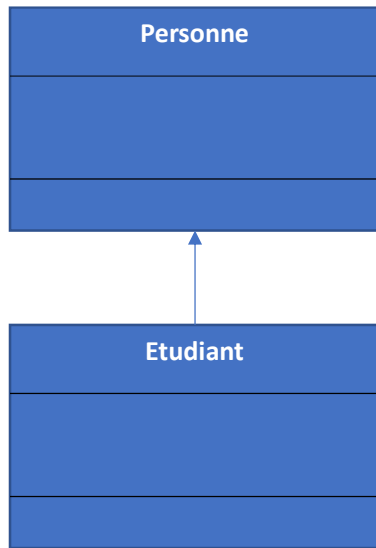
- Une méthode peut être paramétrée par un type, qu'elle soit dans une classe générique ou non.
- L'appel de la méthode nécessite de l'instancier par un type, sauf si le compilateur peut réaliser une inférence de type (déduction des types des données à la compilation)

```
class TableauAlg {  
    public static <T> T getMilieu(T [] tableau) {  
        return tableau[tableau.length / 2];  
    }  
}
```

```
String[] noms= {"Marie", "possède", "une", "petite", "lampe"};  
String milieu = TableauAlg.<String>getMilieu(noms) ;  
Ou String milieu = TableauAlg.getMilieu(noms) ;
```

```
Lion[] noms= {new Lion("Simba"), new Lion(" Laith ")}  
Lion milieu = TableauAlg.<Lion>getMilieu(noms) ;
```

# Règles d'héritage pour les types génériques



« Est-ce que **Paire<Etudiant>** est une sous-classe de **Paire<Personne>** ? ».

**Etudiant[]** etudiants = ... ;

**Paire<Personne>** personne = TableauAlg.minmax(etudiants);



**Généralement** il n'existe pas de relation entre **Paire<S>** et **Paire<T>**, quels que soient les éléments auxquels S et T sont reliés.

# Les types Wildcard (Joker)

on voudrait qu'une méthode écrite pour un type générique T puisse fonctionner avec toute instance des sous-classes de T



Les concepteurs Java ont inventé une Solution : le type Joker

on peut spécifier qu'un paramètre de type est toute sous-classe ou sur-classe d'une classe

- `<?>` désigne un type inconnu
- `<? extends C>` désigne un type inconnu qui est soit C soit un sous-type de C
- `<? super C>` désigne un type inconnu qui est soit C soit un sur-type de C
- C peut être une classe, une interface ou un paramètre de type



# Exemple du type Wildcard (Joker)

**<? extends Personne>**

remplace toute paire générique dont le paramètre TYPE est une sous-classe de Personne : comme Paire<Etudiant> mais pas Paire<String>

```
public static void afficheBinômes(Paire<Personne> personnes) {  
    Personne p = personnes.getFst();  
    Personne d = personnes.getSnd();  
    System.out.println(p.getNom()+" et "+d.getNom()+" sont  
    ensembles."); }
```

```
public static void afficheBinômes(Paire<? extends Personne>  
    personnes) { Personne p = personnes.getFst();  
    Personne d = personnes.getSnd();  
    System.out.println(p.getNom()+" et "+d.getNom()+" sont  
    ensembles."); }
```

**Voici, donc  
une utilisation  
possible :**

```
public class Main  
{ public static void main(String[] args)  
{ Personne personne1 = new Personne("Lagafe", "Gaston");  
    Personne personne2 = new Personne("Talon", "Achile");  
    Paire<Personne> bynômePersonne = new Paire<Personne>(personne1, personne2);  
    Etudiant etudiant1 = new Etudiant("Guillemet", "Virgule");  
    Etudiant etudiant2 = new Etudiant("Mouse", "Mickey");  
    Paire<Etudiant> bynômeEtudiant = new Paire<Etudiant>(etudiant1, etudiant2);  
    afficheBinômes(bynômePersonne);  
    afficheBinômes(bynômeEtudiant); }
```

# Exercices d'application

## Exercice 1.

Repérer les erreurs commises dans les instructions suivantes :

```
class C <T>
{
    T x ;
    T[] t1 ;
    T[] t2 ;
    public static T inf ;
    public static int compte ;
    void f (){
        x = new T () ;
        t2 = t1 ;
        t2 = new T [5] ;
    }
}
```



# Exercices d'application

## Exercice 2. Méthode générique


1. Écrire une méthode générique permettant d'additionner les éléments d'un tableau et de retourner la somme au format double.

Tester la méthode pour des entiers puis pour des doubles.

2. Écrire une méthode générique fournissant en retour un objet tiré au hasard dans un tableau fourni en argument.

Écrire un petit programme utilisant cette méthode.

3. Écrire une méthode qui renvoie au hasard un objet choisi parmi deux objets de même type fournis en argument. Écrire un petit programme utilisant cette méthode.



# Exercices d'application

## Exercice 3. Classe générique


On dispose de la classe générique suivante :

```
class Couple<T>{  
    private T x, y ;  
    // les deux éléments du couple  
    public Couple (T premier, T second){  
        x = premier ;  
        y = second ;  
    }  
    public void affiche (){  
        System.out.println ("premiere valeur : " + x+ " - deuxieme valeur : " + y ) ;  
    }  
}
```



# Exercices d'application

## Travail demandé:

1. Créer, par dérivation, une classe CoupleNomme permettant de manipuler des couples analogues à ceux de la classe Couple<T>, mais possédant, en outre, un nom de type String. On redéfinira convenablement les méthodes de cette nouvelle classe en réutilisant les méthodes de la classe de base.
  2. Toujours par dérivation à partir de Couple<T>, créer cette fois une « classe ordinaire » (c'est-à-dire une classe non générique), nommée PointNomme, dans laquelle les éléments du couple sont de type Integer et le nom, toujours de type String.
  3. Écrire un petit programme de test utilisant ces deux classes CoupleNomme et PointNomme.
- 



## **Partie 2: Les collections**

# Avantages des collections Vs des Tableaux



Les tableaux Java sont simples d'utilisation mais :

- Ne sont pas redimensionnables; il faut spécifier la taille à l'avance
- Si on veut dépasser la taille déclarée, il faut créer un nouveau tableau puis copier l'ancien
- L'insertion d'un élément au milieu oblige à déplacer tous les éléments qui suivent.
- La recherche d'un élément dans un grand tableau est longue lorsqu'il n'est pas trié.
- Les indices pour accéder aux éléments d'un tableau doivent être entiers.
- Les tableaux ne peuvent contenir des éléments de type différent


Le package `java.util`

**Solution**

Plusieurs classes de collection utilisées pour gérer un ensemble d'éléments de type d'objet

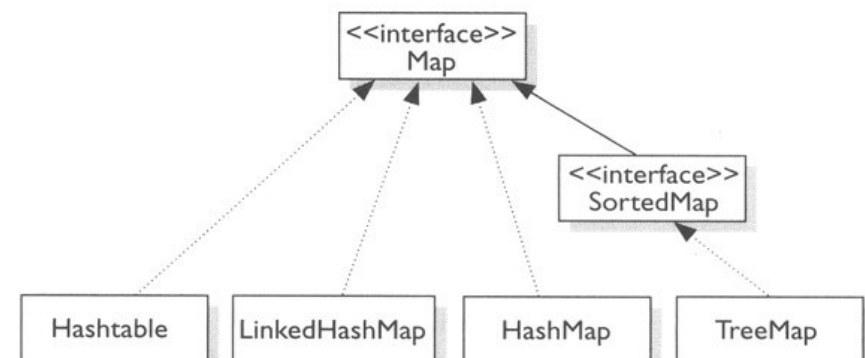
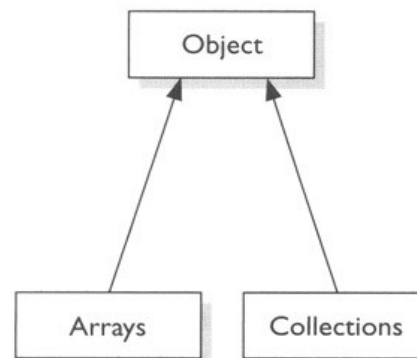
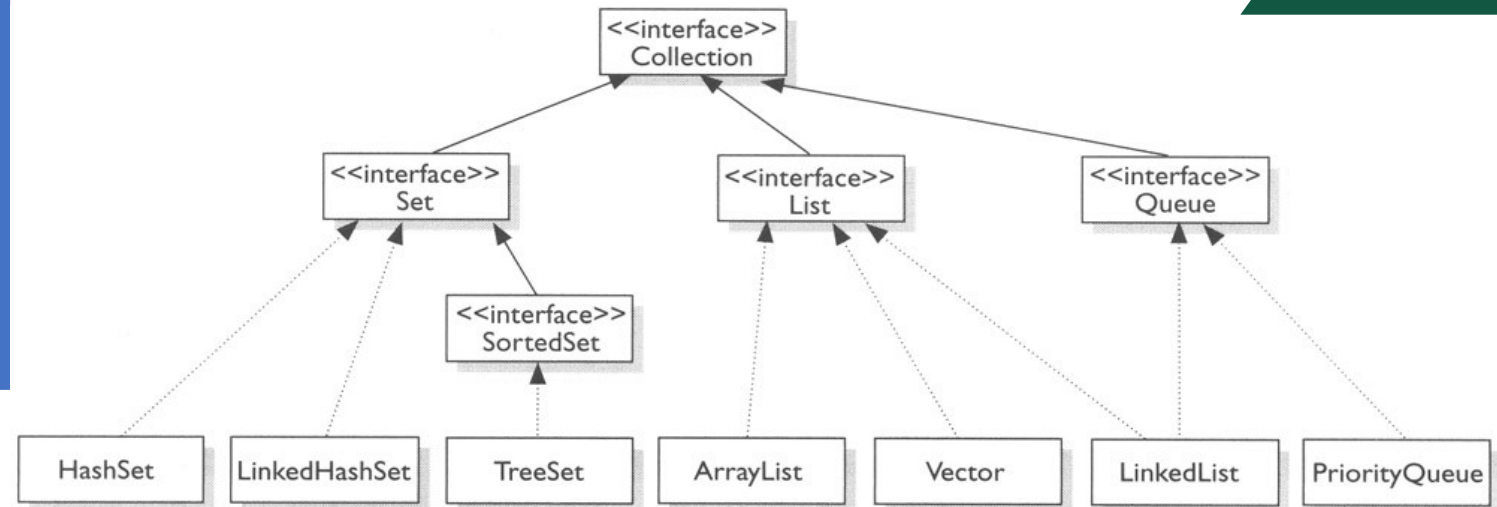
Il y a principalement trois types de collection : les **List**, les **Set** et les **Map**, chaque type a ses avantages et ses inconvénients.

# Avantages des collections Vs des Tableaux

- Les collections gérées par les classes **ArrayList et LinkedList**
  - gèrent des ensembles ordonnés d'éléments accessible par leur indice.
  - peuvent contenir des éléments égaux.
  - Chaque élément mémorisé ayant une position déterminée,  
 Ces classes ressemblent aux tableaux Java mais ne sont pas limitées en taille.
- Les collections gérées par les classes **HashSet et TreeSet**
  - met en œuvre la théorie des ensembles et ne peuvent donc pas contenir des éléments égaux.
  - La classe TreeSet stocke les éléments de son ensemble dans l'ordre ascendant.
- Les collections gérées par les classes **HashMap et TreeMap**
  - utilisent une clé pour accéder aux éléments au lieu d'un indice entier.
  - La classe TreeMap stocke les éléments de sa collection dans l'ordre ascendant des clés.



# Les collections



implements

extends

# L'interface Collection

Collection est la super-interface commune à List et Set. C'est une interface générique.

MÉTHODES DE CONSULTATION	
<b>boolean isEmpty()</b>	retourne vrai ssi la collection est vide.
<b>int size()</b>	retourne le nombre d'éléments contenus dans la collection.
<b>boolean contains(Object e)</b>	retourne vrai ssi la collection contient l'élément donné.
<b>boolean containsAll(Collection&lt;E&gt; c)</b>	retourne vrai ssi la collection contient tous les éléments de la collection donnée
MÉTHODES D'AJOUT	
<b>boolean add(E e): boolean</b>	ajoute l'élément donné à la collection, et retourne vrai ssi la collection a été modifiée
<b>addAll(Collection&lt;E&gt; c)</b>	ajoute à la collection tous les éléments de la collection donnée, et retourne vrai ssi la collection a été modifiée
MÉTHODES DE SUPPRESSION	
<b>void clear()</b>	supprime tous les éléments de la collection
<b>boolean remove(E e)</b>	supprime l'élément donné, s'il se trouve dans la collection
<b>boolean removeAll(Collection&lt;E&gt; c)</b>	supprime tous les éléments de la collection donnée
<b>boolean retainAll(Collection&lt;E&gt; c)</b>	supprime tous les éléments qui ne se trouvent pas dans la collection donnée



# La classe ArrayList

- La classe `ArrayList` est semblable à un tableau de taille dynamique



- Elle est totalement adaptée lorsque nous ajoutons ou supprimons des éléments en fin de tableau.



- La suppression ou bien l'ajout d'un élément au milieu nécessite beaucoup de temps machine,
  - Tous les éléments situés après l'élément supprimé doivent être décalés d'une case.
  - De même pour l'ajout



# La classe LinkedList

Que doit on faire si nous voulons ajouter des éléments au milieu du tableau?



- Il existe donc une classe qui permet de surmonter les points faibles de ARRAYLIST, il s'agit de la classe **LINKEDLIST**.
- Mémorise ses éléments avec une liste doublement chaînée, ce qui permet d'insérer plus rapidement un élément dans la collection,
- Ralentit l'accès à un élément par son indice.

# Les méthodes communes de ArrayList et LinkedList

Méthodes	Caractéristiques Communes
boolean <b>add</b> (T élément) void <b>add</b> (int indice, T élément)	Ajoute un élément soit à un indice donné soit en fin de collection.
Void <b>clear</b> ( ) void <b>remove</b> (int indice) void <b>remove</b> (T élément)	Suppression de tout ou partie des éléments de la collection.
T <b>get</b> (int indice ) void <b>set</b> (int indice, T élément)	Interrogation et modification d'un élément de la collection à un indice donné.
boolean <b>contains</b> (T élément) int <b>indexOf</b> (T élément) int <b>lastIndexOf</b> (T élément)	Recherche d'un élément dans la collection, en utilisant la méthode equals() pour comparer les objets.
int <b>size</b> ( )	Taille de la collection.

# Les méthodes de ArrayList

Méthodes	java.util.ArrayList
ArrayList<T> () ArrayList<T> (int taille)	Construit un tableau vide avec une spécification de capacité ou pas.
void <b>ensureCapacity</b> (int taille)	Vérifie que le tableau a la capacité nécessaire pour contenir le nombre d'éléments donné, sans nécessiter la réallocation de son tableau de stockage interne.
void <b>trimToSize</b> ()	Réduit la capacité de stockage du tableau à sa taille actuelle.
boolean <b>isEmpty</b> ()	Détermine si le tableau est vide ou pas.

# Les méthodes de LinkedList

Méthodes	java.util.LinkedList
LinkedList<T>()	Construit une liste vide.
void <b>addFirst</b> (T élément) void <b>addLast</b> (T élément)	Ajoute un élément au début ou à la fin d'une liste.
T <b>getFirst</b> () T <b>getLast</b> ()	Renvoie l'élément situé au début ou à la fin d'une liste.
T <b>removeFirst</b> () T <b>removeLast</b> ()	Supprime et renvoie l'élément situé au début ou à la fin d'une liste.

# Exemple – Array List

**Exemple 1.** On veut parcourir un array list.

```
public class Main {  
    public static void main (String []args) {  
        ArrayList liste =new ArrayList();  
        liste.add(0);  
        liste.add(1);liste.add(2);  
        liste.add(3);  
        for(Object elt: liste) {  
            if(elt.equals(0))  
                liste.remove(elt);  
        }  
    }  
}
```

Le résultat est l'exception suivante:

```
Exception in thread "main" java.util.ConcurrentModificationException at  
java.base/java.util.ArrayList$Itr.checkForComodification(ArrayList.java:1009)  
at java.base/java.util.ArrayList$Itr.next(ArrayList.java:963)  
at org.eclipse.test.Main.main(Main.java:15)
```



**Solution**

Les itérateurs



# L'interface *Iterator*

- Une interface permettant d'énumérer les éléments contenus dans une collection.
- Toutes les collections proposent une méthode itérateur renvoyant un itérateur.
- Parcours dans un sens uniquement.
- Pour les `ArrayList`, une boucle `for` contenant un `get(i)` est une mauvaise idée.
- On utilise alors une boucle `for-each` avec un itérateur.

L'interface *Iterator* contient les méthodes suivantes :

- **`hasNext()`**: retourne `true` si l'itérateur contient d'autres éléments
- **`next()`**: retourne l'élément suivant de l'itérateur
- **`remove()`**: supprime le dernier objet obtenu par `next()`

```
1 List<String> l = ...;  
2 Iterator<String> i = l.iterator();  
3 while (i.hasNext()) {  
4     String s = i.next();  
5     System.out.println(s);  
6 }
```

# Solution proposée du problème

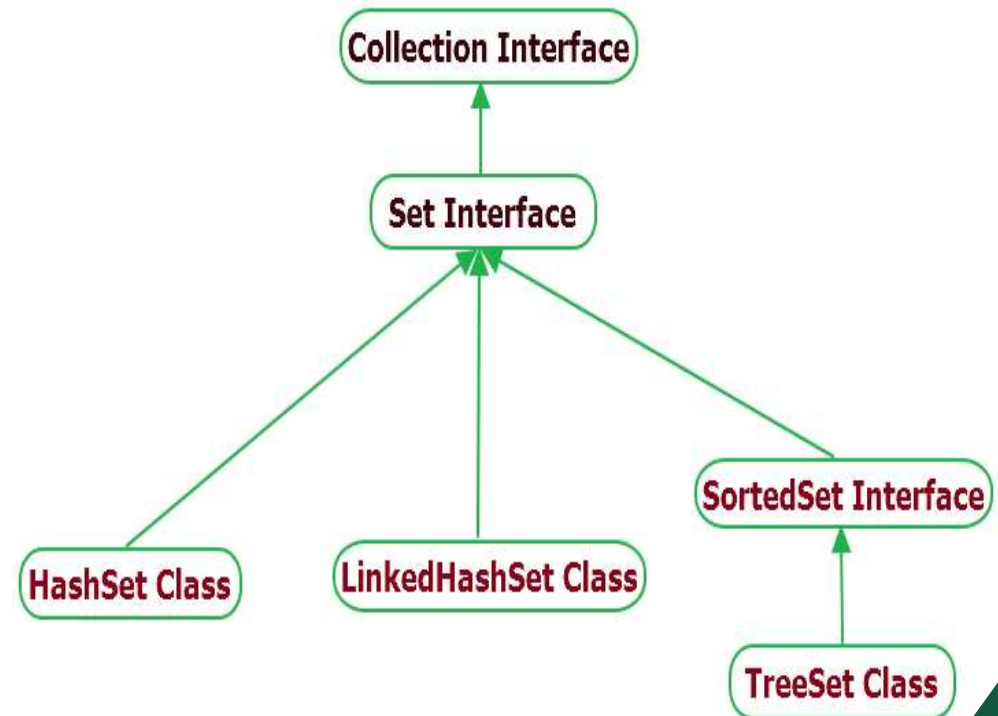
Réolvons le problème précédent avec les itérateurs

```
package org.eclipse.classes;
import java.util.ArrayList;
import java.util.Listlterator;
public class Main {
    public static void main(String[] args) {
        ArrayList<Integer> liste =newArrayList();
        liste.add(0); liste.add(1); liste.add(2); liste.add(3);
        Listlterator<Integer> li = liste.listlterator();
        while(li.hasNext()) {
            Integer elt = li.next();
            if(elt.equals(0)) li.remove();}
            System.out.println(liste);
            //affiche[1,2,3]
        }
    }
```



# Interface Set


- Un *Set* est une collection qui ne contient pas d'éléments dupliqués.
  - La méthode `equals()` est utilisée pour comparer les éléments.
- ➔ il peut n'y avoir qu'un seul élément qui vaut *null*.
- Elle représente un ensemble dans le sens mathématique
  - « **Collection d'objets distincts, non ordonnés** »
    - **HashSet** mémorise ses éléments dans un ordre quelconque (et donc plus rapide si l'ordre n'a pas d'importance),
    - **TreeSet** mémorise ses éléments dans l'ordre ascendant avec un arbre, pour maintenir plus rapidement le tri des éléments stockées.





# Les méthodes communes à HashSet et TreeSet

méthodes	Caractéristiques Communes
boolean <b>add</b> (T élément)	Ajout d'un élément dans la collection si l'objet n'est pas déjà dans la collection.
void <b>clear</b> ( ) void <b>remove</b> (T élément)	Suppression de tout ou partie des éléments de la collection.
boolean <b>contains</b> (T élément)	Recherche d'un élément dans la collection, en utilisant la méthode equals() pour comparer les objets.
int <b>size</b> ( )	Taille de la collection.



# Les méthodes de HashSet et TreeSet

méthodes	java.util.HashSet
HashSet<T>()	Construit une collection vide.
méthodes	java.util.TreeSet
TreeSet<T>()	Construit une collection vide.
TreeSet<T>( Comparator<? super O > c )	Construit une collection vide à partir d'un objet qui implémente l'interface Comparator afin de spécifier comment trier la classe T.
T <b>first</b> () T <b>last</b> ()	Renvoie l'élément situé au début ou à la fin d'une liste en sachant que les éléments de cette collection sont dans l'ordre.
boolean <b>isEmpty</b> ()	Détermine si la collection est vide ou pas.



# Exemple - HashSet

```
package org.eclipse.classes;
import java.util.HashSet;
import java.util.Iterator;
public class Main {
    public static void main(String[] args) {
        HashSet hs = new HashSet();
        hs.add("bonjour"); hs.add(69); hs.add('c');
        Iterator it = hs.iterator();
        while(it.hasNext()) System.out.println(it.next());
    }
}
```

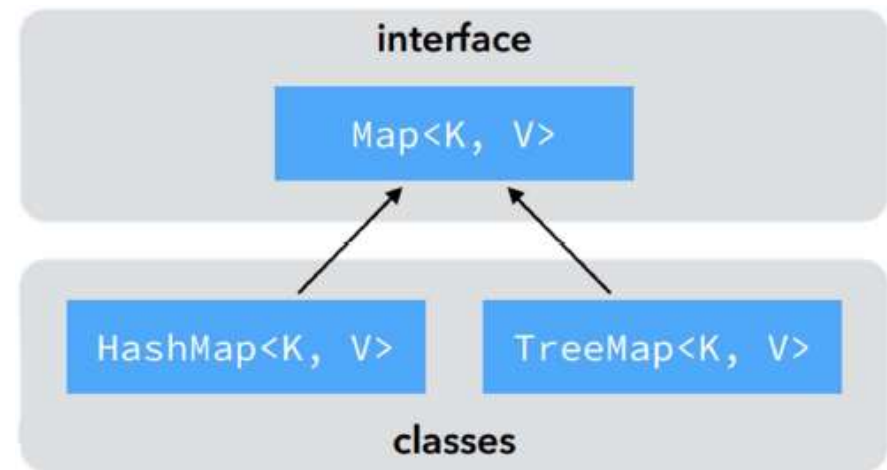
**Résultat d'exécution**

```
c
69
bonjour
```



# Tables associatives : Map

- Un objet de type Map stocke des couples clef-valeur. On parle aussi de dictionnaire.
- La clef doit être unique mais une valeur peut avoir plusieurs clefs.
- L'interface **Map<K,V>** fixe les méthodes pour manipuler de tels couples. Les clefs sont de type K et sont associées aux valeurs de type V.



# Les classes HashMap et TreeMap

- *Gèrent des collections d'éléments accessibles par une clé correspondant à un élément. Ces classes mémorisent en fait un ensemble d'entrées associant une clé et son élément (key-value).*
- *L'accès par clé dans une collection est comparable à l'accès par indice dans un tableau :*

## Exemple:

- vous pouvez disposer d'un répertoire téléphonique dans lequel les clés sont les noms des personnes et les valeurs les numéros de téléphone.
- *HashMap* mémorise ses entrées dans un ordre quelconque,
- *TreeMap* mémorise ses entrées dans l'ordre ascendant avec un arbre, pour maintenir plus rapidement le tri des éléments stockés.



# Les méthodes communes à HashMap et TreeMap

méthodes	Caractéristiques Communes
V <b>get</b> (K clé) V <b>put</b> (K clé ,V valeur)	Interrogation et modification d'un élément donné de la collection avec une clé.
void <b>clear</b> ( ) void <b>remove</b> (K clé)	Suppression de tout ou partie des éléments de la collection.
boolean <b>containsKey</b> (K clé) boolean <b>containsValue</b> (V valeur)	Recherche d'un élément dans la collection par sa clé ou sa valeur. Ces méthodes utilisent la méthode equals() pour comparer clé aux clés ou valeur aux valeurs de la collection.
int <b>size</b> ( )	Taille de la collection.
Set<K> <b>keySet</b> ( )	Retourne l'ensemble des clés.
Collection<K> <b>values</b> ( )	Retourne l'ensemble des valeurs.
Set<Map.Entry<K, V>> <b>entrySet</b> ()	Retourne l'ensemble des paires clé / valeur.

# Les méthodes de HashMap et TreeMap

Méthodes	java.util.HashMap
HashMap<K, V >()	Construit un tableau vide avec une spécification de capacité ou pas.
boolean isEmpty()	Détermine si la collection est vide ou pas.
méthodes	java.util.TreeMap
TreeMap<K, V>()	Construit une liste vide.
TreeMap<T>( Comparator<? super O > c )	Construit une collection vide à partir d'un objet qui implémente l'interface Comparator afin de spécifier comment trier la classe T.
K firstKey () K lastKey ()	Retourne la première ou la dernière clé en sachant que les éléments de cette collection sont dans l'ordre.

# Exemple - HashMap

```
package org.eclipse.classes;
import java.util.HashMap; import java.util.Set;
Public class Main{public static void main(String[] args){
    HashMap<Integer, String> hm =new HashMap();
    hm.put(1, "Java");
    hm.put(2, "PHP");
    hm.put(10, "C++");
    hm.put(17,null);
    Set s = hm.entrySet();
    Iterator it = s.iterator();    while(it.hasNext())
        System.out.println(it.next());
    }
    // Pour afficher la clé et la valeur, on peut utiliser l'Entry
    for(Entry<Integer, String> entry : hm.entrySet()) {
        System.out.println(entry.getKey() + " " +
        entry.getValue  ());
    }
}
```

# Comparaison d'objets: L'interface Comparable



Comment *TreeSet* (ou *TreeMap*) sait-il dans quel ordre trier les éléments ?



Il faut que les objets manipulés implémentent l'interface Comparable ce qui est d'ailleurs le cas pour *String*



Que se passe t-il si nous essayons de stocker des objets qui n'implémentent pas cette interface ?



une erreur d'exécution (Exception levée).

## Interface Comparable

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

L'appel *a.compareTo(b)* doit renvoyer *0* si *a* et *b* sont égaux, un nombre entier négatif si *a* est inférieur à *b* et un nombre entier positif si *a* est supérieur à *b*.

# Comparaison d'objets: L'interface Comparable

```
public class Liste {  
    public static void main(String[] args) {  
        TreeSet<Personne> ensemble = new TreeSet<Personne>();
```

```
    }  
  
    class Personne implements Comparable<Personne> {  
        private String nom, prénom;  
        private int âge;  
        public Personne(String nom, String prénom, int âge)  
        { this.nom = nom; this.prénom = prénom; this.âge = âge; }  
        public String getNom() { return nom; }  
        public String getPrénom() { return prénom; }  
        public int getÂge() { return âge; }  
        public int compareTo(Personne p) {  
            if (!nom.equalsIgnoreCase(p.nom))  
                return nom.compareToIgnoreCase(p.nom);  
            if (!prénom.equalsIgnoreCase(p.prénom))  
                return prénom.compareToIgnoreCase(p.prénom);  
            return âge - p.âge; }  
    }
```

# Comparaison d'objets: L'interface Comparator



Il arrive que des données doivent être comparées selon plusieurs critères.  
Exemple comparer des personnes selon leurs âges et selon leurs poids.



Dans ce cas, associer à la classe un ordre naturel en lui faisant implémenter l'interface Comparable n'est pas suffisant. L'API offre une autre interface, Comparator.



***Il faut faire passer dans le constructeur TreeSet (ou MapSet) un objet Comparator***

## Interface Comparator

```
public interface Comparator<T> { int compare(T a, T b); }
```

***La méthode compare() renvoie une valeur entière négative si a est inférieure à b, nulle si a et b sont identiques, et positive dans le dernier cas.***

# Comparaison d'objets: L'interface Comparator

```
class TrierPersonneAge implements Comparator<Personne> {  
    public int compare(Personne p1, Personne p2) {  
        return p1.getÂge() - p2.getÂge(); } }  
}
```

```
class TrierPersonnePoids implements Comparator<Personne> {  
    public int compare(Personne p1, Personne p2) {  
        return p1.getPoids() - p2.getPoids(); } }  
}
```

```
public class Liste {  
    public static void main(String[] args) {  
        Personne schmidt, lagafe;  
        TreeSet<Personne> ensembleAge = new TreeSet<Personne>(new TrierPersonneAge());  
        TreeSet<Personne> ensemblePoids = new TreeSet<Personne>(new TrierPersonnePoids());  
    }  
}
```





## Exemple Tri d'une liste d'entiers

```
import java.util.*;  
public class Coll1  
{ public static void main(String args[])  
{ List liste = new ArrayList();  
liste.add( new Integer(9));  
liste.add(new Integer(12));  
liste.add(new Integer(5));  
Collections.sort(liste);  
Iterator it = liste.iterator();  
while (it.hasNext())  
{ System.out.println((it.next()).toString());  
} } }
```



# Exemple Tri d'une liste de personne



```
import java.util.*  
public class ListePersonneTest  
{ public static void main(String[] args)  
{ List liste = new ArrayList();  
  Personne p1 = new Personne("Xyel","Pierre");  
  Personne p2 = new Personne("Durand","jean");  
  Personne p3 = new Personne("Smith","Joe");  
  liste.add(p1);  
  liste.add(p2);  
  liste.add(p3);  
  Collections.sort(liste, new ComparatorPersonneNom()) } }
```

# Les collections et les types génériques

En utilisant les génériques, on peut écrire notre classe ArrayList de la façon suivante:

```
public class ArrayList<T>
{ public T get() { ... }
  public void add(T t) { ... } }
```

Dans cette écriture, T est le type générique. Lors de l'instanciation de cette classe, on doit lui donner une valeur, qui doit être une classe ou une interface. Dans le cas où T représente le type String, notre classe devient :

```
public class ArrayList<String>
{ public String get() { ... }
  public void add(String s) { ... } }
```

# Les collections et les types génériques

Voyons un exemple de déclaration d'une telle variable.

```
ArrayList<String> listOfString = new ArrayList<>() ;  
ArrayList<Integer> listOfInteger = new ArrayList<>() ;
```

**Des Questions ?**

