

# M2103 – Bases de la Programmation Orientée Objets



## Java – Cours 9

### Collections

# Plan du Cours

---

- **Introduction**
- Interfaces Collection et Bases pour l'Implémentation
- Généricité
- Listes & Implémentation
- Ensembles & Implémentation
- Classes d'Enveloppement et Conversion de Types
- Itérateurs et Boucle *for* Optimisée
- Maps & Implémentation
- Comparaison de Données
- Méthodes Utilitaires pour les Collections

# Collections

---

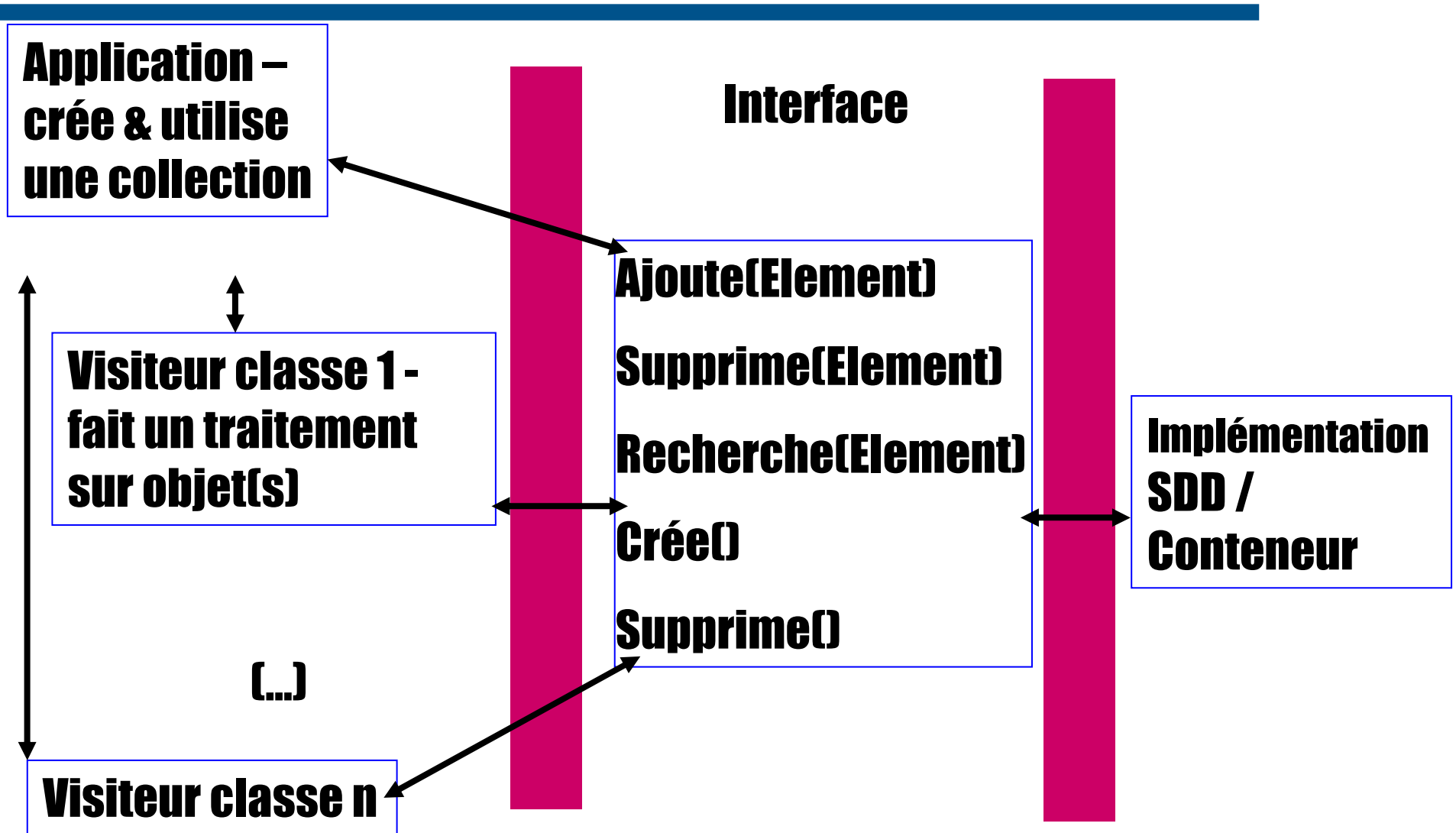
- **Définition**

- Spécification d'un ensemble de données avec les opérations sur celles-ci pour la gestion de ces données
- Cette abstraction permet aux utilisateurs de la collection d'ignorer les détails de l'implémentation relatifs aux données/algorithmes, et de se concentrer sur les principes de son utilisation. Ceci favorise la réutilisabilité.

- **Exemples**

- Liste, Pile, File, Dictionnaire...

# Collections



# Collections Java (Java Collections Framework)

---

- Trois composants
  - **Interfaces (types)** : Interfaces Java ; *description* de différents types abstraits de collections
  - **Implémentations** : Classes Java ; *implémentation* du type abstrait
  - **Algorithmes** : méthodes pour la *manipulation* des collections; disponibles pour différents types de collections
- Avantages
  - Moins d'efforts de programmation
  - Augmente la vitesse et la qualité de l'écriture de code
  - Réutilisation du code

# Plan du Cours

---

- Introduction
- **Interfaces Collection et Bases pour l'Implémentation**
- Généricité
- Listes & Implémentation
- Ensembles & Implémentation
- Classes d'Enveloppement et Conversion de Types
- Itérateurs et Boucle *for* Optimisée
- Maps & Implémentation
- Comparaison de Données
- Méthodes Utilitaires pour les Collections

# Interfaces Collections

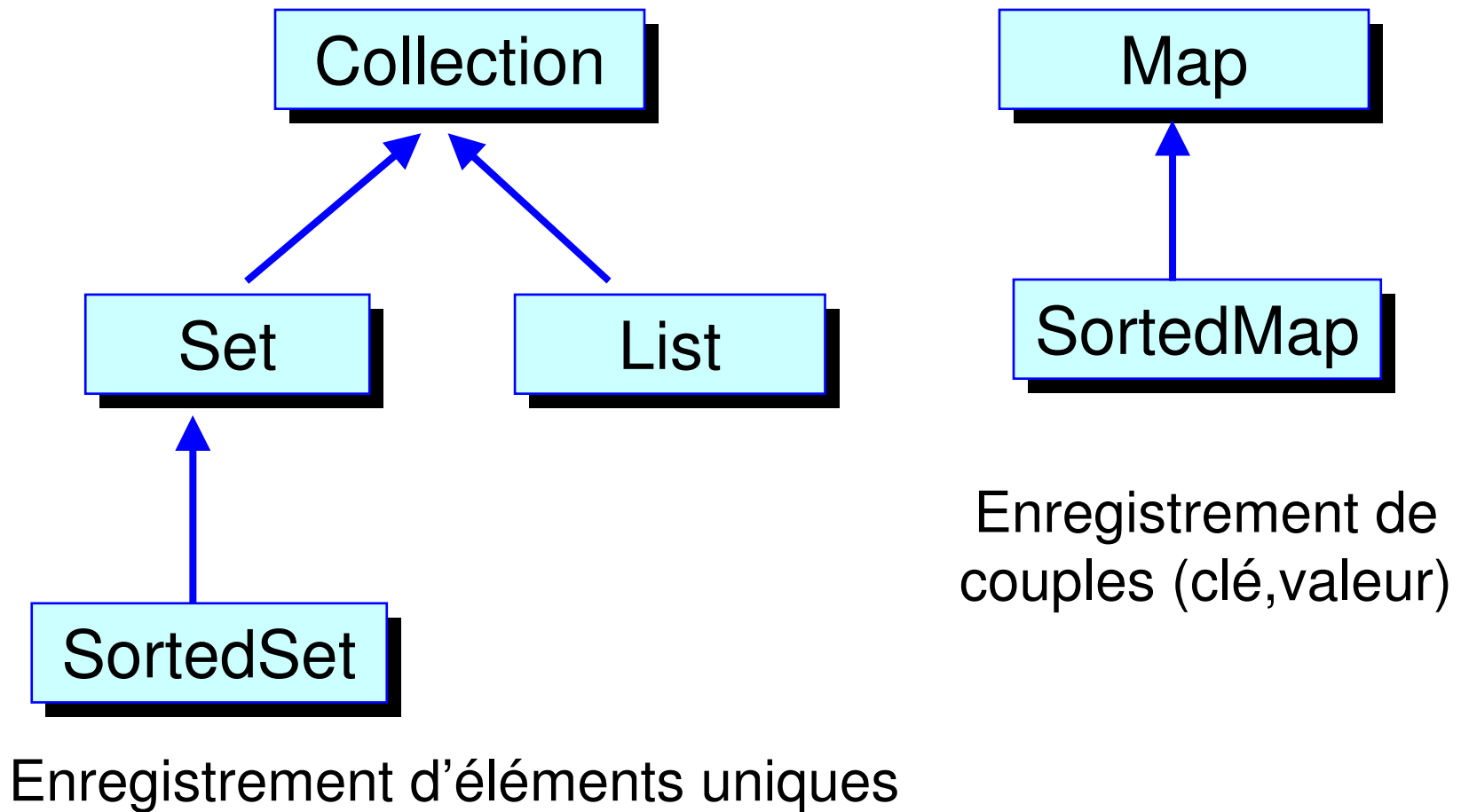
---

- Vues abstraites des collections
- Focus sur la fonctionnalité
- Ne présentent pas les détails liés à l'implémentation

Généralement, lorsqu'on met en oeuvre une collection, l'interface est choisie en premier

# Interfaces Collections de Base

---





# Interfaces Collections de Base

---

## ■ **List**

- Position (séquence); permet les éléments dupliqués; accès par position
- **ArrayList**, **LinkedList**
- *Exemple* : classement d'une course

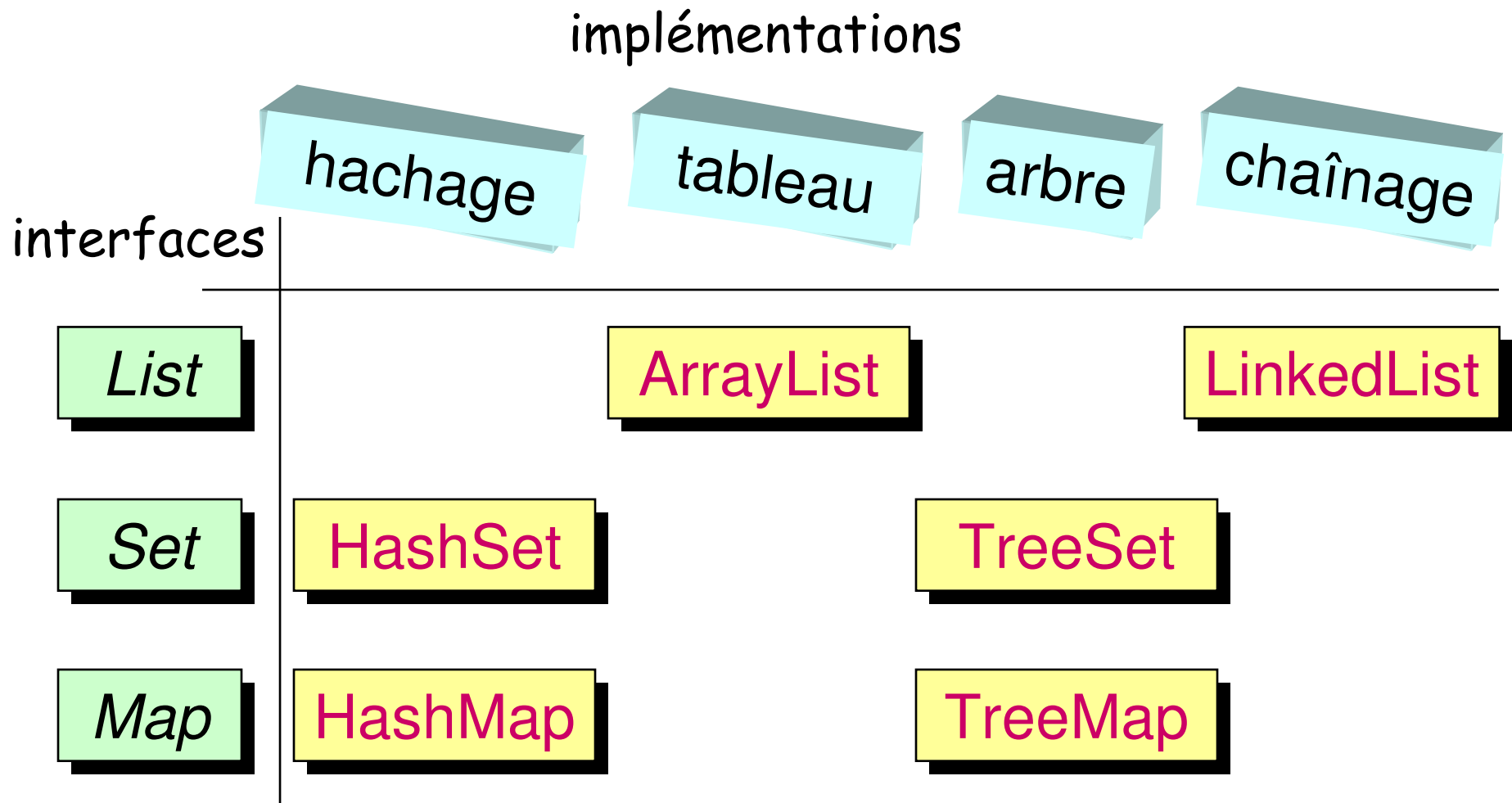
## ■ **Set**

- Pas d'éléments dupliqués; pas de position
- **HashSet**, **TreeSet**
- *Exemple* : étudiants dans une classe

## ■ **Map**

- Paires clé-valeur stockés (clés uniques); accès par clé
- **HashMap**, **TreeMap**
- *Exemple* : annuaire

# Implémentation



# Plan du Cours

---

- Introduction
- Interfaces Collection et Bases pour l'Implémentation
- **Généricité**
- Listes & Implémentation
- Ensembles & Implémentation
- Classes d'Enveloppement et Conversion de Types
- Itérateurs et Boucle *for* Optimisée
- Maps & Implémentation
- Comparaison de Données
- Méthodes Utilitaires pour les Collections

# Méthodes de l'Interface Collection

---

**Collection :**  
super-type  
abstrait de  
toutes les  
collections

```
public interface Collection<E>
{
    // Méthodes de base
    int size();
    boolean isEmpty();
    boolean contains(Object o);
    boolean add(E o);
    boolean remove(Object o);
    Iterator<E> iterator();
    (...)
}
```

# Programmation Générique

---

- *Java 5.0* a introduit un mécanisme nommé “**Généricité**” pour permettre aux programmeurs de spécifier le type d’une **Collection** au moment de la compilation, de manière à ce que le compilateur puisse réaliser une vérification de type, pour éviter que des objets d’un type non voulu soient placés dans la collection à l’exécution.
- Cela permet d’éviter de faire un *typecast* pour les objets dans des méthodes utilisant les collections.

Eg.

```
public void uneMethode(Collection<Personne> c)
{
    for (Iterator i = c.iterator(); i.hasNext(); )
        if (i.next().estEtudiant())
            // traitement Etudiant
    ...
}
```

*Nous verrons ce qu’est  
un “iterator” dans la suite*

# Utilisation (étapes typiquement requises)

---

- Choix de l'interface (la fonctionnalité requise), eg. **List**
- Choix de l'implémentation (classe), eg. **LinkedList**
- Déclaration d'une instance de la Collection en utilisant le type Interface, eg. **List<Personne> maCollection;**
- Création d'une instance de la Collection en utilisant le type pour l'implémentation, et affectation à l'instance de type Interface, eg.  
**maCollection = new LinkedList<Personne>();**
- Utilisation de la collection via les méthodes de l'interface, eg. **maCollection.add(unePersonne);**

# Plan du Cours

---

- Introduction
- Interfaces Collection et Bases pour l'Implémentation
- Généricité
- **Listes & Implémentation**
- Ensembles & Implémentation
- Classes d'Enveloppement et Conversion de Types
- Itérateurs et Boucle *for* Optimisée
- Maps & Implémentation
- Comparaison de Données
- Méthodes Utilitaires pour les Collections

# List

---

- Choix de **List** si :
  - Collection maintient des éléments avec une position mais non triée
  - Ajout et suppression d'objets facilités
  - Objets dupliqués sauvegardés
- Après le choix de l'interface **List**, 2 implémentations possibles
  - **ArrayList<E>** pour l'accès aux éléments en temps constant par indice
  - **LinkedList<E>** pour l'ajout et la suppression d'éléments en maintenant les positions, peu d'accès



# ArrayList VS LinkedList

---

- ArrayList
  - Accès par indice rapide
  - Insertion et suppression 'lentes'
  - Performance
    - Ajustement taille - coûteux
    - Lenteur caractérisée pour un grand nombre d'éléments
- LinkedList
  - Accès séquentiel
  - Insertion et suppression plus 'rapides'
  - Performance
    - Lent pour l'accès par position

# Exemple

---

```
public class ClasseA
{
    List <Chose> maCollectionListe;           // Liste de "Choses"
    . . .
    public ClasseA()
    {
        maCollectionListe = new ArrayList<Chose>();    // est 1 ArrayList
                // ou
        maCollectionListe = new LinkedList<Chose>();   // est 1 liste chaînée
    }
}
```

- Puis utilisation des méthodes de l'interface **List** pour manipuler la collection

# Avantages de l'Approche

---

Exemple: utilisation de `LinkedList` pour une `List`

```
LinkedList<Chose> choses;  
choses = new LinkedList<Chose> ();  
traite(choses);  
  
void traite(LinkedList<Chose> liste)  
{  
    ...  
}
```

# Changement en ArrayList

---

```
ArrayList<Chose> choses;  
choses = new ArrayList<Chose>();  
traite(choses);
```

```
void traite(ArrayList<Chose> liste)  
{  
    ...  
}
```



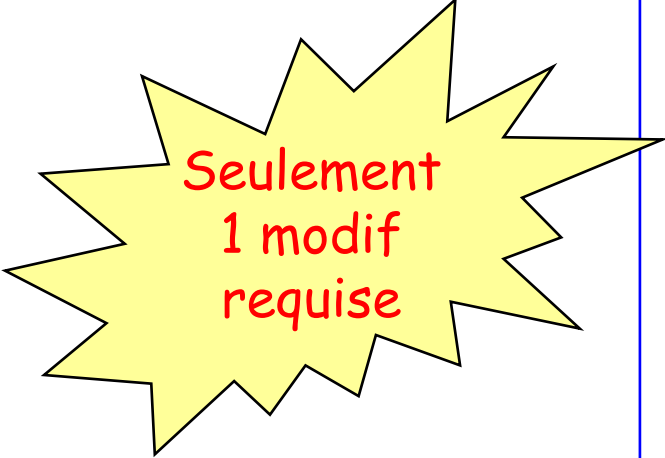
3 modifs  
déjà  
requises

En changeant la liste chaînée en `ArrayList`, chaque occurrence de `LinkedList` doit être remplacée par `ArrayList` (déclarations de variables et paramètres, instantiation d'objet).

# Meilleures Implémentations

**Solution** : Utilisation d'interfaces (ici `List`) pour les déclarations. Utilisation de la classe d'implémentation une fois seulement pour l'instanciation de l'objet.

```
List<Chose> choses = new LinkedList<Chose> ();  
                    //new ArrayList<Chose> ();  
traite(choses);  
  
void traite(List<Chose> liste)  
{  
    ...  
}
```



Seulement  
1 modif  
requis

# Plan du Cours

---

- Introduction
- Interfaces Collection et Bases pour l'Implémentation
- Généricité
- Listes & Implémentation
- **Ensembles & Implémentation**
- Classes d'Enveloppement et Conversion de Types
- Itérateurs et Boucle *for* Optimisée
- Maps & Implémentation
- Comparaison de Données
- Méthodes Utilitaires pour les Collections

# Set

---

- Choix de l'interface **Set** si :
  - Elements maintenus dans la collection sans position
  - Insertion et suppression facilitées
  - Rejet d'objets dupliqués
- Après le choix de l'interface **Set**, 2 implémentations possibles
  - **HashSet<E>** pour l'accès fréquent (implémentation permettant la manipulation la plus rapide, cf. tables de hachage)
  - **TreeSet<E>** pour retourner des éléments triés

# Exemple

---

```
public class ClasseA
{
    Set<Chose> maCollectionEnsemble;
    ...
    public Classe1()
    {
        maCollectionEnsemble = new HashSet<Chose>();
        // ou
        maCollectionEnsemble = new TreeSet<Chose>();
    }
}
```



# Plan du Cours

---

- Types de Données Abstraits & Collections : Introduction
- Interfaces Collection et Bases pour l'Implémentation
- Généricité
- Listes & Implémentation
- Ensembles & Implémentation
- **Classes d'Enveloppement et Conversion de Types**
- Itérateurs et Boucle *for* Optimisée
- Maps & Implémentation
- Comparaison de Données
- Méthodes Utilitaires pour les Collections

# Exemple : Types de Base et Collections

---

Insertion d'un entier dans un **ArrayList**?

- Éléments d'un **ArrayList** sont des **Objects**
- **int** n'est pas un sous-type de **Object**
- Un **int** ne peut être inséré dans un **ArrayList**!
- Solution ?.....

# Conversion Types de Base en Types Classe

## Autoboxing

---

Processus automatique après jdk 5.0 : **Autoboxing**

// Valeur de type **float** automatiquement convertie en  
// un objet **Float**

```
Float f = 234.567F; // Pas d'erreur
```

// De manière similaire

```
float value = 234.567F;  
Float f = value; // Pas d'erreur
```

# Classes d'Enveloppement

---

- Integer, Long, Float, Double, Short, Byte, Character, Boolean
- Toute classe d'enveloppement a 2 types de constructeurs
- Récupère en entrée un élément dont le type est un type de base et retourne un objet correspondant
  - **Integer(5)** retourne un objet **Integer** de valeur 5
- Récupère en entrée une chaîne de caractères et retourne un objet correspondant
  - **Integer("45")** retourne un objet **Integer** de valeur 45
  - **Character** est la seule classe d'enveloppement qui ne présente pas ce constructeur

# Exemple : ArrayList d'Entiers

---

int converti en un objet Integer avant d'être ajouté à ArrayList

```
ArrayList <Integer> listeEntiers;  
listeEntiers = new ArrayList<Integer>();  
int nombre = 42;  
listeEntiers.add(nombre);  
...
```

Après extraction de Integer à partir de ArrayList, conversion automatique en int à partir de Integer

```
Integer unEntier;  
unEntier = listeEntiers.get(2);  
int monNombre = unEntier;
```

# Plan du Cours

---

- Introduction
- Interfaces Collection et Bases pour l'Implémentation
- Généricité
- Listes & Implémentation
- Ensembles & Implémentation
- Classes d'Enveloppement et Conversion de Types
- **Itérateurs et Boucle *for* Optimisée**
- Maps & Implémentation
- Comparaison de Données
- Méthodes Utilitaires pour les Collections

# Itérateurs

Itérateurs utilisés pour le parcours des collections : visite de chaque objet

Retourne un objet Iterator pour une collection

Vérifie s'il y a d'autres éléments

Retourne l'objet suivant

Supprime le dernier objet retourné par next()

```
public interface Collection<E>
{
    ...
    Iterator<E> iterator();
}

public interface Iterator<E>
{
    boolean hasNext();
    E next();
    void remove();
}
```

# Quand Utiliser un Itérateur au lieu d'un *For* ?

---

- La boucle **for** traditionnelle peut toujours être utilisée pour manipuler des collections, toutefois un ***itérateur*** peut s'avérer plus pratique/moins risqué, eg. :
  - Suppression d'éléments
  - Recherche/remplacement d'éléments
  - Parcours de multiples collections
- Pas d'indices requis!



# Utilisation des Itérateurs

---

Parcours typique d'une collection :

```
List<String> liste1;  
liste1 = new LinkedList<String>(); // ou ArrayList()  
  
public void parcours()  
{  
    Iterator<String> liste1Iterateur = liste1.iterator();  
    while (liste1Iterateur.hasNext())  
    {  
        String temp = liste1Iterateur.next();  
        //traite l'élément.....  
    }  
}
```

# Boucle *For* Optimisée (ou JDK 5)

---

- Utilisée à la place d'un itérateur pour rendre le code plus lisible.

- Eg.

```
List<String> liste1;  
liste1 = new ArrayList <String>();  
// ici code pour ajouter des éléments dans liste1....  
  
for (String s : liste1)  
// signification : “pour chaque String s dans la List liste1, ...”  
{  
    System.out.println(s);  
    // ou toute autre opération...  
}
```

# Plan du Cours

---

- Introduction
- Interfaces Collection et Bases pour l'Implémentation
- Généricité
- Listes & Implémentation
- Ensembles & Implémentation
- Classes d'Enveloppement et Conversion de Types
- Itérateurs et Boucle *for* Optimisée
- **Maps & Implémentation**
- Comparaison de Données
- Méthodes Utilitaires pour les Collections

# Implémentations de l'Interface `Map`

---

- **HashMap**

- Stocke des couples (clé, valeur)
  - Exemple : (Nom, Identifiant)
- Utilise le hachage des clés pour permettre l'accès rapide aux éléments.

- **TreeMap**

- Même principe que le *HashMap* sauf que les éléments sont retournés triés.

# Quelques méthodes de Map (1/3)

---

- Ajout d'éléments:

**put(C cle, V valeur)**

```
Map<String, String> map1 =  
    new HashMap<String, String> ();  
Map<String, Integer> map2 =  
    new TreeMap<String, String> ();  
.  
. .  
for (String s: noms)  
{  
    map1.put(s, identifiant1);  
    map2.put(s, identifiant1);  
}
```

## Quelques méthodes de `Map` (2/3)

---

- Récupérer une valeur
  - **`typeValeur get(Object cle)`**
  - Retourne la valeur couplée à *cle*

```
String iD = map1.get("Joe");  
  
System.out.println("Identifiant étudiant de Joe  
est "  
  
+ iD);
```

## Quelques méthodes de Map (3/3)

---

- Récupérer les clés par **keySet()** retourne un **Set**
- Récupérer les valeurs par **values()** retourne une collection (Set ou List)

```
Set<String> mesCles = new HashSet<String>();  
Collection<String> mesValeurs = new HashSet<String>();  
mesCles = map1.keySet();  
mesValeurs = map1.values();
```

# Plan du Cours

---

- Introduction
- Interfaces Collection et Bases pour l'Implémentation
- Généricité
- Listes & Implémentation
- Ensembles & Implémentation
- Classes d'Enveloppement et Conversion de Types
- Itérateurs et Boucle *for* Optimisée
- Maps & Implémentation
- **Comparaison de Données**
- Méthodes Utilitaires pour les Collections



# Utilisation de `TreeSet` et `TreeMap`

---

- Ces collections retournent des éléments ordonnés.
- Pour que ceci puisse fonctionner, les éléments doivent pouvoir être comparés entre eux.
- Pour les classes implémentant l'interface **`Comparable`**, il n'y a rien à faire, eg. **`String`**
- Mais comment comparer deux **`Etudiants`** (classe définie par l'utilisateur)?

# Comparaison d'Autres Objets

---

- L'interface **Comparable<T>** fournit la méthode ***compareTo***  
**int compareTo(T obj)**
  - Compare l'objet courant avec l'objet spécifié.
  - Retourne une valeur entière indiquant le résultat de la comparaison.

- Processus

- Etape 1 : Classe implémente l'interface **Comparable<T>**

e.g.

```
public class Etudiant implements Comparable<Etudiant>
{ ... }
```

- Etape 2 : Ecriture de l'implémentation de la méthode ***compareTo***

```
public int compareTo(Etudiant e)
{
    return nom.compareTo(e.nom);
}
```

# Alternative

---

- Problèmes :
  - Classe n'implémente pas l'interface **Comparable**
  - Implémentation de la comparaison non désirée, eg. utilisation de l'*identifiant* au lieu du *nom*
  - Besoin d'ordonner selon différents attributs
- Solution : Un comparateur sur mesure
  - Un comparateur est un objet implémentant l'interface **Comparator** et permettant de définir une méthode de comparaison sur mesure

# Exemple

---

```
public class EtudiantCompareur implements Comparator<Etudiant>
{
    public int compare(Etudiant e1, Etudiant e2)
    {
        String nom1 = e1.getName();
        String nom2 = e2.getName();
        return nom1.compareTo(nom2);
    }
}
```

# Exemple

---

```
EtudiantCompareur eC = new EtudiantCompareur();  
Set<Etudiant> sE = new TreeSet<Etudiant>(eC);
```

- Crée un `TreeSet` basé sur le compareur.
- Possibilité de créer autant d'objets compareurs que voulu.
- Note sur `TreeMap`
  - Le type *clé* doit implémenter **`Comparable`** ou utiliser un compareur.
  - Bien sûr, aucune contrainte sur le type *valeur*.

# Plan du Cours

---

- Introduction
- Interfaces Collection et Bases pour l'Implémentation
- Généricité
- Listes & Implémentation
- Ensembles & Implémentation
- Classes d'Enveloppement et Conversion de Types
- Itérateurs et Boucle *for* Optimisée
- Maps & Implémentation
- Comparaison de Données
- **Méthodes Utilitaires pour les Collections**

# Méthodes Utilitaires pour les Collections

---

- La classe `Collections` présente des méthodes statiques pour :
  - Chercher et trier
  - Inverser l'ordre des éléments
  - Copier...
- Exemple : Récupérer la valeur maximale d'une collection  
`List<Integer> myIntegers = ...;`  
`...`  
`Integer maxInt = Collections.max(myIntegers);`